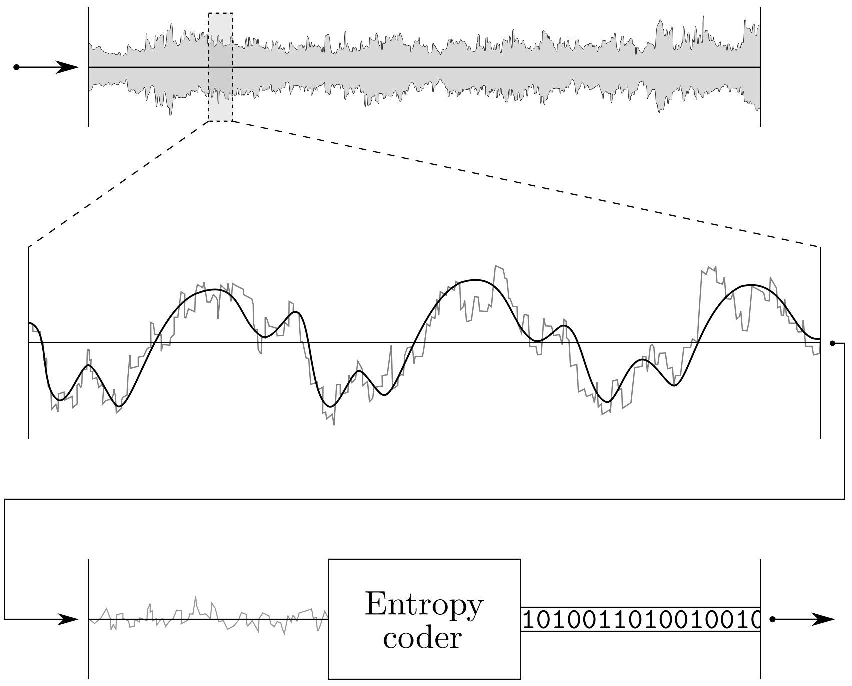
Linear predictive coding and Golomb-Rice codes in the FLAC lossless audio compression codec

Maxim van den Berg

July 2, 2020

Bachelor thesis Mathematics and Computer Science

Supervisors: dr. Taco Walstra, dr. Michael Walter, Freek Witteveen



Informatics Institute
Korteweg-de Vries Institute for Mathematics
Faculty of Sciences
University of Amsterdam

Abstract

The Free Lossless Audio Codec (FLAC) is the most widely used format for lossless digital audio compression. Most current lossless audio compression algorithms, including FLAC, consists of the same three fundamental parts. Namely, compression is realized by partitioning of the audio signal, the modelling of each section individually using linear predictive coding, and the efficient encoding of the model error using a suitable entropy coder. We present three standard linear prediction algorithms in detail, namely the Levinson-Durbin algorithm, the Burg algorithm and the covariance Burg algorithm, and introduce a new adaptive approach based on covariance Burg which seeks to optimise the partitioning of the audio signal in order to increase the effectiveness of linear prediction. Furthermore, we compare the standard and new algorithms in terms of compression ratio and compression time when integrated into the FLAC format specification. Although we found compression ratios of the adaptive algorithms to be similar to the standard algorithms, the relatively efficient blocksize detection shows promise for further work. Additionally, we discuss the general Golomb codes, a popular class of entropy coders, and provide a theoretical demonstration of optimality for Golomb-Rice codes for the encoding of the linear prediction residuals specifically.

Title: Linear predictive coding and Golomb-Rice codes in the FLAC lossless audio compression codec
Author: Maxim van den Berg, maximvdberg@gmail.com, 11867000
Supervisors: dr. Taco Walstra, dr. Michael Walter, Freek Witteveen
Second graders: A. van Inge, dr. Jan Brandts
End date: July 2, 2020

Informatics Institute
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
http://www.ivi.uva.nl

Korteweg-de Vries Institute for Mathematics
University of Amsterdam
Science Park 904, 1098 XH Amsterdam
http://www.kdvi.uva.nl

Contents

1 An introduction to lossless audio compression

1.1 Lossless audio compression

Compression, the process of representing digital data in a storage-efficient manner by detecting and removing any inherent redundancies contained in it, is fundamental in the disciple of digital audio. Although continued developments of storage space and internet bandwidth have decreased its explicit necessity, the need for good audio compression is still substantial, be it for large-scale audio repositories —such as for scientific or archival purposes–, in the countless regions with poor internet and WiFi infrastructures, or for costly audio network streaming. And of course, one should remember that smaller file sizes result in less network traffic for all scenarios, which is beneficial for both power consumption and, in turn, the environment. Audio compression therefore serves both practical and ethical purposes, motivating the need for its constant improvement as other technologies develop alongside it.

Compression algorithms, and audio compression algorithms specifically, can be divided in two main categories: *lossy* compression algorithms, where loss of non-essential data is encouraged to facilitate better compression, and *lossless* compression, where no loss of data is permitted. With lossy compression, the reconstructed data is only an approximation of what was originally compressed, however with lossless compression the original data can be reconstructed in its entirety.

Lossy formats, such as the well-known (MP3) and the open source Ogg Vorbis, are predominant in everyday use, since many applications do not require the perfect detail which the lossless formats provide: for audio compression, the small errors matter little in the final listening experience for most casual listeners. There is however always an important trade-off between the quality of the decoded approximation and the compression ratio. Our focus will be on lossless formats, thus it is important to realise we need not concern ourselves with this trade-off: lossless compression requires that the original quality is maintained perfectly. Lossless is therefore generally preferred for music production/performance settings, archival purposes, medical applications and scientific settings [21]. Three key factors are then as follows:

- **Compression ratio:** defined as the compressed file size —in bits— divided by the original file size. For lossless compression, lower ratios are better.
- **Encoding speed:** the average time it takes to compress a file of a given size. Shorter is favorable, although since encoding is typically done just once per audio file, in most situations one can afford slightly longer encoding times.
- **Decoding speed:** the average time it takes to reconstruct the audio signal from the given compressed data. For music particularly, this indicates how long it will take decode the audio file *every* time it is listened to. For most application, it is essential for decompression to be quick.

The latter two are slightly imprecise as they depend on the hardware involved, but better defined measures exists to allow for meaningful statements across formats.

In general, lower compression ratios allow for increased encoding and decoding speeds, and conversely, shorter encoding and decoding times naturally lead to higher compression ratios. This trade-off is crucial, and advancements in the field frequently present an improvement in either of these three factors.

Lossless compression schemes traditionally rely on the same fundamental techniques, as is the case for the open source Free Lossless Audio Codec (FLAC) [1] —currently the most widely used format for lossless audio— , the Apple Lossless Audio Codec (ALAC), the MPEG-4 Audio Lossless Coding (ALS) [16], the more recent IEEE Advanced Audio Coding standard [12] and the older Shorten [24] —on which most of the listed formats are based. The newer formats, such as ALS, usually provide improvements on the basis principles, but do not deviate considerably [16]. There is one notable exception, the MPEG-4 Scalable to Lossless (SLS) format, which seeks to combine support for lossless as well as lossy decompression within the same file [28]. It can be described better as an extension of a common, and fundamentally different, approach which is used mainly for lossy compression. We will discuss it briefly in the last section of the introduction.

We will use FLAC as the basis for our discussion, since it is by far the most popular lossless codec, completely open source, and uses, among other things, a basic form of the general lossless compression technique mentioned before. In order to introduce it properly, we begin with a short introduction to digital audio signals.

1.2 Digital audio signals

It is common to write an *analog audio signal* as a function $f\colon \mathbb{R} \to \mathbb{R}$ which maps a time variable t to the pressure $f(t)$ at time t. Unfortunately, the finiteness of computer arithmetic forces us to make a countable approximation of f when converting to the digital realm. Fortunately however, this can be done —as far as human perception is concerned— losslessly. The connection between the original and encoded signal is based on two terms:

- The *sampling frequency* $f_s \in \mathbb{N}$, in samples per second, and
- The total duration T of the signal, in seconds.
  (note that it is perfectly allowed for T to be infinite).

By the Nyquist-Shannon sampling theorem, if all frequencies in the signal are lower than $\frac{1}{2}f_s$, the complete signal can be reconstructed from its digital representation [17].

Definition 1.1 We define a *digital audio signal* as a sequence

$$\big\{X_n\big\}_{\substack{n\in\mathbb{N}\\ n<m}} \subset \mathbb{D}, \text{ where}$$

$$\begin{cases} \mathbb{D} & \text{is the numeric domain, e. g. } \big\{x \in \mathbb{Z}\colon \log|x| \leq 31\big\} \\ m := \lfloor f_s \cdot T \rfloor & \text{is the length of the sequence} \end{cases}$$

Note that there are in fact multiple ways audio signals can be represented, and which one is preferred depends on the theoretical and practical context. It is not uncommon, for instance, to write $\big\{X_n\big\}$ as a vector $X \in \mathbb{D}^m$ when working with concepts from linear algebra, or as a function $x\colon \mathbb{N} \to \mathbb{D}$, $x[n] := X_n$ when certain functional transformations are required. We will use the appropriate notation for each application, so one should be familiar with all of them.

Additionally, audio often comes in multiple channels. When this is the case we write $X_n^{(c)}$ with $c \in C$ to distinguish between them. C a collection of *channels*, such as the channels $\{\texttt{left}, \texttt{right}\}$ in the case of stereo.

We will now disregard the samplerate and the total length of the audio signal, and focus exclusively on the digital representation, where all compression will take place.

1.3 A high-level overview of FLAC

As was mentioned before, the FLAC encoding algorithm exhibits traits general to most lossless audio standards. They are summarized as the following step, and a visual outline can be found in figure 1.3.

1. **The blocking step**, where the digital audio stream is split into blocks of a certain length N, and each chunk is encoded independently. N is typically around 4096 samples long for a sample rate of 44100 (about a sixth of a second).
2. **The linear prediction step**, where the signal within a block is approximated by a *linear predictive model*, in which every sample is predicted based on the preceding samples according to a static set coefficients. Efficiently finding the optimal coefficients for this model is central, and this is what most distinguishes different compression algorithms. This step is referred to as linear predictive coding (LPC).
3. **The entropy coding step**, where the remaining errors of the prediction model, referred to as the *residuals*, are encoded losslessly and efficiently using an entropy encoding scheme. These codes can work either sample-per-sample or on multiple residuals at once.

The data can then be communicated as the linear prediction coefficients together with the encoded residuals, from which the original signal can be reconstructed. The LPC model —or any model for that matter— will never perfectly describe the audio signal. However, we can guarantee that the residuals will generally be small relative to the original signal, such that the entropy coding step to be done effectively and the data is compressed adequately.

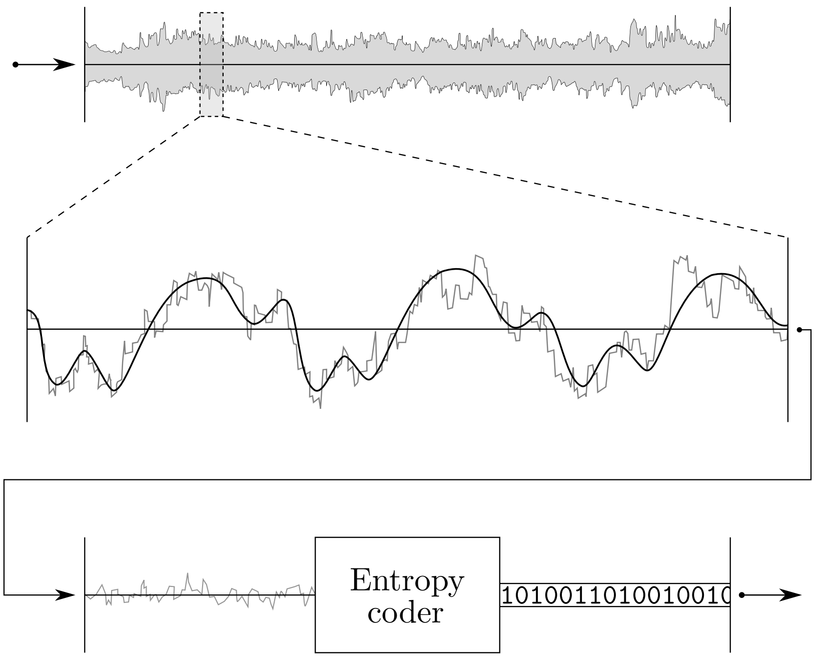


Figure 1.1: A visual representation of the three steps as summarized above. The audio is split up into small blocks, each of which is approximated with linear prediction, and the model error is encoded efficiently with some entropy coder.

The choice of linear prediction as the central component has become specific to lossless formats. Lossy compression codecs, such as ALS, typically use a more direct frequency representation than linear prediction, such as the *modified discrete cosine transform* (MDCT) which is closely related to the discrete Fourier transform. This does provide more control over the frequency content in the encoded signal, but this control is unnecessary for lossless compression which disallows the altering of frequency content.

FLAC defines an explicit and strict format for the encoded data and a reference encoder is provided, but in principle the methods for encoding are not fixed. The codec consists of the following components:

- A stereo signal will often be encoded as a `mid` and `side` channel, the average and difference channels respectively, during what is called the *interchannel decorrelation step*, such that

  $$X_n^{(\texttt{mid})} := \frac{X_n^{(\texttt{left})} + X_n^{(\texttt{right})}}{2} \quad \text{and} \quad X_n^{(\texttt{side})} := X_n^{(\texttt{left})} - X_n^{(\texttt{right})}.$$

  Since the `left` and `right` channels are likely to be heavily correlated, this immediately significantly reduces the combined size of the two channels.

- In the reference encoder, the audio blocks have a fixed blocksize across the whole audio file —by default of size 4096—, although the format allows for variable blocksizes.

- The linear prediction coding is computed using the *Levinson-Durbin algorithm.* However, FLAC allows for additional models —a constant value and a (fixed) prediction polynomial— as well.

- The entropy encoder with which the residuals are encoded is the *Golomb-Rice code*, which require an additional parameter estimated according to the residuals.

In the coming two chapters we will illustrate the described main steps which the lossless audio formats share, after which we will provide a more detailed discussion on FLAC itself in chapter 4. There we provide proposals for extension of both the standard FLAC encoder as well as the format specification itself. In chapter 2 we will analyse linear predictive coding, providing two classic formulations and algorithms, namely the standard Levinson-Durbin algorithm and the lattice Burg algorithm, as well as a proposal for a new algorithm based on Burg which can efficiently recognize optimal blocksizes concurrently to the computation of the model coefficients. In chapter 4, all methods are compared. Chapter 3 will be concerned with the entropy encoder used by FLAC, and the more general class of Golomb codes is described. Moreover, a proof for the optimality of the Golomb-Rice is given, based on residuals distributed following the discrete 2-sided geometric distribution.

2 Linear predictive coding

Linear predictive coding, abbreviated as LPC, is one of the oldest and widely applied audio modelling techniques in audio coding. It finds its origin in speech encoding specifically, were it was used to efficiently encode speech for communication over low-bandwidth digital channels [21, 18]. At its core, a LPC model predicts any sample of an audio signal s as a linear combination of the preceding samples. More precisely, for a coefficient set $\{a_j\}_{1\leq j\leq p}$ it makes a prediction $\hat{s}[n]$ for $s[n]$ by calculating

$$\hat{s}[n] = \sum_{j=1}^{p} a_j s[n-j].$$

As we will see, this can be done effectively for signals with *static* spectral contents, as is the case for speech formants.

Although the preceding interest in predictive coding of audio signals had existed for some time, the first proposals for the now standard techniques where made by F. Itakura and S. Saito in 1966. The methods were subsequently developed further by them, J. Burg, J. Makhoul and others, eventually leading to its application in *Voice-Over-IP*, or VOIP, protocols in 1974 on the internet's predecessor ARPANET[11] [13].

It is perhaps no surprise that LPC found its way into applications concerning file compression as well. Although the main concepts were designed with speech in mind, LPC lends itself to general audio compression without requiring extensions. Most lossless standards rely on LPC as the foundation for the compression to this day, as can be found in Shorten by T. Robinson [24], on which FLAC is based, or the MPEG-4 Audio Lossless Coding[16]. Lossy compression standards on the other hand typically avoid LPC and choose instead for more explicit spectral modeling techniques such as the (modified) discrete cosine transform, which provide more control of the spectral detail necessary for lossy encoding [2].

In this chapter we will take an in-depth look at LPC and concern ourselves with different methods for computing the optimal model parameters. We begin with a formal derivation of the general concept, for which we require the introduction of some mathematical background first. Afterwards we describe the basic idea more in-depth, and discuss three ways of computation: the standard *linear least squares* approach based on a straightforward autoregressive filter model, an alternative method using so-called *lattice filters* and an extension to allow for instantaneous per-sample updating of the LPC parameters. Finally, we will discuss some properties of the *residuals* —the difference between the original signal and the predicted model— which will be important for efficient lossless compression. A comparison of compression performance between the algorithms is provided in chapter 4.

2.1 Mathematical background - The $\mathcal{Z}$-domain

Throughout this section and the next, we will consider a *digital signal* to be a function $s\colon \mathbb{N}\cup\{0\} \to \mathbb{D}$, although for our purposes the exact details of $\mathbb{D}$ are not relevant as the concepts are general.

In its most general sense, the $\mathcal{Z}$-transform is a way to convert between a series of values — which we will refer to as the *time-domain*— and their (complex) *frequency-domain* representation. Namely, for any such series it defines an unique function on the complex plane.

Definition 2.1 ($\mathcal{Z}$-transform) Given a bounded discrete-time function $s\colon \mathbb{Z} \to \mathbb{D}$ we define its *$\mathcal{Z}$-transform* $S\colon \mathbb{C} \to \mathbb{C}$ as

$$S(z) := \sum_{n=-\infty}^{\infty} s[n] z^{-n}.$$

We can then extend this definition to digital signals, or sequences in general, by setting $s[n] = 0$ for $n \in \mathbb{Z} \setminus \mathbb{N}$.

The corresponding *region of convergence* (ROC) R_S is then defined as the set for which $S(z)$ converges, namely as

$$R_S = \Big\{ z \in \mathbb{C} \mid \big|S(z)\big| < \infty \Big\}.$$

We write $s[n] \overset{\mathcal{Z}}{\to} S(z)$, and in general we will stick to this lower/upper case convention.

For signals of which the unit circle $\big\{ z \in \mathbb{C} \mid |z| = 1 \big\} = \big\{ e^{i\omega} \mid \omega \in [-\pi, \pi) \big\}$ is contained in R_S there is a natural connection with the *Discrete-time Fourier transform.* Specifically, we find the following.

Proposition 2.2 Given a discrete-time function $s\colon \mathbb{Z} \to \mathbb{D}$ and $s \overset{\mathcal{Z}}{\to} S$ for which the unit circle $\big\{ e^{i\omega} \mid \omega \in [-\pi, \pi) \big\} \subseteq R_S$, the *Discrete-time Fourier transform* is given by

$$S\big(e^{i\omega}\big) = \sum_{n=-\infty}^{\infty} s[n] e^{-i\omega n}.$$

The corresponding *inverse transform* then equals

$$s[n] = \frac{1}{2\pi} \int_{-\pi}^{\pi} S\big(e^{i\omega}\big) \cdot e^{i\omega n} \, \mathrm{d}\omega.$$

The inverse transform can be generalized to any $\mathcal{Z}$-transformed function S by a contour integral through the ROC R_S which encircles the origin, however we will omit the details as we do not require it explicitly. For our purposes, the properties in the next theorem will suffice.

Theorem 2.3 Let $x \overset{\mathcal{Z}}{\to} X$, $y \overset{\mathcal{Z}}{\to} Y$ and $a_1, a_2 \in \mathbb{D}$, then the following holds:

		Time-domain		Frequency-domain	ROC
(a)	**Linearity:**	$a_1 x[n] + a_2 y[n]$	$\overset{\mathcal{Z}}{\to}$	$a_1 X(z) + a_2 Y(z)$	Contains $R_X \cap R_Y$
(b)	**Time shift:**	$x[n-k]$	$\overset{\mathcal{Z}}{\to}$	$z^{-k} X(z)$	Can lose 0 if $k > 0$
(c)	**Convolution:**	$(x * y)[n]$	$\overset{\mathcal{Z}}{\to}$	$X(z)Y(z)$	Contains $R_X \cap R_Y$

Where $(s * h)[n] := \displaystyle\sum_{k=-\infty}^{\infty} s[n] \cdot h[n-k]$.

The main advantage of the $\mathcal{Z}$-transformation lies in the fact that many operations concerning frequency and phase content reduce to simple multiplications in the $\mathcal{Z}$-domain, which can then be translated to the relatively efficient operations on the original discrete signal using the previous theorem. This leads to the concept of the *digital filters.* Given a discrete signal s we can define a *filter* H such that multiplication of S by H transforms the frequency content in a desirable fashion, while making sure this operation can be brought back to the time-domain in an efficient manner. To achieve the latter, H will generally be chosen to be a *rational function.*

Definition 2.4 (Digital filters) A *rational filter* H, defined as a *rational transfer function*, is of the form

$$H(z) = \frac{P\big(1/z\big)}{Q\big(1/z\big)}$$

where P and Q are both polynomials with real coefficients and of finite degree, say ℓ and p respectively. Often, the constant coefficient of Q is divided away and fixed to 1 to remove redundancy, such that

$$H(z) = \frac{P\left({}^1\!/\!_z\right)}{Q\left({}^1\!/\!_z\right)} = \frac{\sum_{k=0}^{\ell} b_k z^{-k}}{1 + \sum_{k=1}^{p} a_k z^{-k}}$$

Applying this filter to a signal s is then done by multiplication in the $\mathcal{Z}$-domain. Calling the resulting signal u, we find that

$$\begin{aligned} U(z) = S(z)H(z) = S(z)\frac{P\left({}^1\!/\!_z\right)}{Q\left({}^1\!/\!_z\right)} \\ U(z)Q\left({}^1\!/\!_z\right) = S(z)P\left({}^1\!/\!_z\right) \end{aligned}$$

Now, note that $Q({}^1\!/\!_z)$ and $P({}^1\!/\!_z)$ are the $\mathcal{Z}$-transformation of their respective coefficient vectors q and p. As a direct consequence, theorem 2.3 allows us to rewrite this filter into a recursive formula in the time-domain, namely as

$$\begin{aligned} (u * a)[n] = (s * b)[n] \\ u[n] + \sum_{k=1}^{p} a_k u[n-k] = \sum_{k=0}^{\ell} b_k s[n-k]. \end{aligned}$$

Note that $u[n]$ can be calculated as a linear combination of past and present values of the input signal s, as well as past values of u. Transfer functions for which this is the case are referred to as *causal* transfer functions, and it comes as no surprise these type of filters are favored for most applications. Moreover, filters for which $p = 0$ are generally referred to as *moving average* (MA) filters, as $u[n]$ becomes a weighted average of the last $\ell + 1$ values of s. When instead $\ell = 0$, the filter is called *autoregressive* (AR). Filters of the described form, which are both AR and MA, are therefore generally referred to as ARMA filters.

Filters are often characterized by the *poles* and *zeros* of their transfer function.

Definition 2.5 For a digital filter $H(z) = P({}^1\!/\!_z)/Q({}^1\!/\!_z)$, we call $\zeta \in \mathbb{C}$ a *zero* when $P(\zeta) = 0$. Analogously, we call $\rho \in \mathbb{C}$ a *pole* when $Q(\rho) = 0$.

Formally, a zero ζ —and analogously poles— have an associated degree as well, defined as the highest n such that $P(x) = G(x)(x - \zeta)^n$ for some other polynomial G.

Note that when a point is both a zero and a pole, they can be divided away against each other in H such that only one (or neither) remains.

Loosely speaking, if a frequency on the unit circle (see proposition 2.2) lies close to a pole, it will become more prevalent in the resulting signal. Similarly, if a frequency lies close to a zero it will be filtered out of the signal.

Example 2.6 Consider a real and discrete signal s with unknown frequency content in a range of $[0.\pi)$ and suppose we want to filter out frequencies around $1/4\pi$ and $3/4\pi$. We can then choose a filter H with zeros (of degree 1) such that

$$P\big(e^{\pm 1/4\pi i}\big) = P\big(e^{\pm 3/4\pi i}\big) = 0$$

and poles (of the same degree) close to these zeros such that the effect on surrounding frequencies is minimized. Say for $\alpha \in [1/2, 1)$ such that

$$Q\big(\alpha e^{\pm 1/4\pi i}\big) = Q\big(\alpha e^{\pm 3/4\pi i}\big) = 0$$

Note that the $\pm$ arises due to the symmetry of the Fourier transform on a real signal, although this is immediately advantageous for our filter as it ensures P and Q will have real coefficients. H then becomes

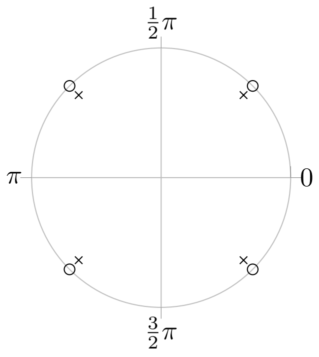


Figure 2.1: The *pole-zero plot* of H, where $\times$ indicates a pole and $\circ$ indicates a zero.

$$H(z) = \frac{(z - e^{1/4\pi i})(z - e^{-1/4\pi i})(z - e^{3/4\pi i})(z - e^{-3/4\pi i})}{(z - \alpha e^{1/4\pi i})(z - \alpha e^{-1/4\pi i})(z - \alpha e^{3/4\pi i})(z - \alpha e^{-3/4\pi i})} = \frac{z^4 + 1}{z^4 + \alpha^4}.$$

The *frequency response* of filter is plotted below, which shows how H reshapes S at the unit disc after multiplication, and thus how the frequency content of s is subsequently changed.

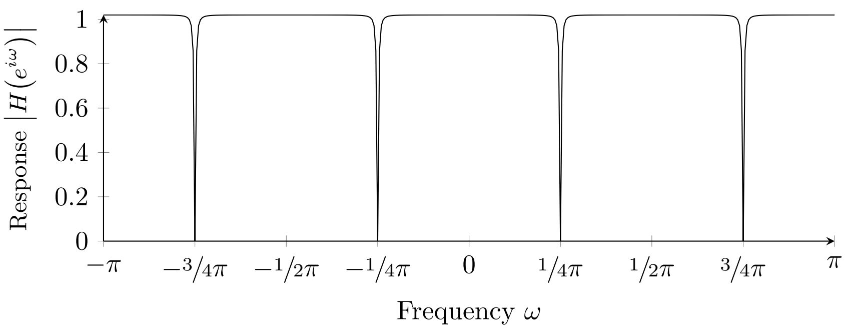


Figure 2.2: The frequency response $\big|H\big(e^{i\omega}\big)\big|$ of the filter H with $\alpha = 0.99$.

This is of course a somewhat artificial example. Real-world signals have frequencies often specified in Hertz, and a corresponding *sampling rate* which translate the continuous time real-world signal to the discrete digital realm. However, this example exhibits some important properties of digital filters, most importantly that the frequency response is only an approximation of what we set out to achieve. In general, the trade-off for implementability causes rational filters always be continuous (apart from the poles), and therefore in many applications never "perfect".

Additionally, filters are often visualised with similar plots as the ones provided: the *pole-zero plot*, which shows the poles and zeros of H in the complex plane, and the *frequency response*, which shows the absolute values of H around the unit disc.

Finally, we shortly state an important property of filters.

Definition 2.7 We call a signal $s\colon \mathbb{Z} \to \mathbb{D}$ *bounded* when there exists a $B \in \mathbb{R}$ such that $\big|s[n]\big| < B$ for all $n \in \mathbb{Z}$.

A transfer function is called *bounded-input bounded-output stable*, or simply *stable*, if for any bounded input signal s, the output signal u, such that $U(z) = S(z)H(z)$, is bounded as well.

If a system and its inverse $Q(^1\!/\!_z)/P(^1\!/\!_z)$ are both stable and causal, we refer to it as a *minimum-phase* system. Digital filters and their inverses are causal by construction, thus we will mostly concern ourselves with this class of systems specifically. An useful characterization for stability is given by the following theorem.

Theorem 2.8 A transfer function H is stable if and only if all poles of H lie within the open unit disc $\{z \in \mathbb{C} \mid |z| < 1\}$.

We conclude this section with a couple of definitions we will be using in the remainder of the chapter. First, we define the notion of stationary processes, which have a close relation to signals that can be effectively described by digital filters. For this we consider our signal s to be, instead of a deterministic set of values, a *stochastic process*, where each individual sample $s[n]$ is a random variable following some unknown distribution.

Definition 2.9 Stochastic process $s\colon \mathbb{Z} \to \mathbb{D}$ is called *stationary* if for each $n, t \in \mathbb{N}$ we have

$$\mathbb{E}\big(s[n+t]\big) = \mathbb{E}\big(s[n]\big) \quad \text{and} \quad \operatorname{Var}\big(s[n+t]\big) = \operatorname{Var}\big(s[n]\big).$$

In particular, any set of sinus waves with fixed respective frequencies is an important example of a stationary process. We will not give a formal proof, but intuitively their associated *phases* —the starting point of the sinusoid— can be considered uniformly random, such that the above properties hold.

Since filter coefficients are static over the complete signal, prediction of the signal can be done effectively only when no large fluctuations of expectancy and variance exists as time progresses. As we will see, stationarity is a significant consideration for linear predictive coding. In the following sections we will use the term “stationary signals” analogously with signals that have unknown but static spectral content, such that additionally the set of frequencies is finite. This is an apparent refinement of the definition —one that better matches digital audio signals— however for our purposes it is one that more directly relates to the effectiveness of linear prediction, which we will use in the next section.

Two important measures for digital signals which we will require are the autocorrelation and covariance.

Definition 2.10 For a given signal $s\colon \mathbb{Z} \to \mathbb{D}$ we call $\mathbb{E}\big(s[n]s[n-i]\big)$ the *autocorrelation* of delay $i \in \mathbb{N} \cup \{0\}$. For a blocksize N, we define the *autocorrelation vector R* as

$$R(i) \coloneqq \sum_{n=i}^{N-1} s[n]s[n-i].$$

We give a simplified definition of the covariance, where the conventional expression is reduced by presuming a zero mean signal, from

$$\mathbb{E}\big[\big(s[n-\ell] - \mathbb{E}(s[n-\ell])\big)\big(s[n-r] - \mathbb{E}(s[n-r])\big)\big] \quad \text{to} \quad \mathbb{E}\big(s[n-l]s[n-r]\big).$$

This is not a big assumption for audio signals, which are typically normalized around 0.

Definition 2.11 For a given signal $s\colon \mathbb{Z} \to \mathbb{D}$ with mean 0, we call $\mathbb{E}\big(s[n-l]s[n-r]\big)$ the *covariance* of delay $r, l \in \mathbb{N} \cup \{0\}$. For any signal s and blocksize N, we define the *covariance matrix* φ as

$$\varphi(\ell, r) := \sum_{n=\max\{\ell,r\}}^{N-1} s[n-\ell]s[n-r].$$

Note that $\varphi(i, 0) = \varphi(0, i) = R(i)$. Moreover, it is possible to calculate the covariance matrix recursively from the autocorrelation vector by the fact that

$$\begin{aligned}\varphi(\ell+1, r+1) = \sum_{n=\max\{\ell+1,r+1\}}^{N-1} s[n-\ell-1]s[n-r-1] &= \sum_{n=\max\{\ell,r\}}^{N-2} s[n-\ell]s[n-r] \\ &= \varphi(\ell, r) - s[N-1-\ell]s[N-1-r].\end{aligned}$$

The given vector R and matrix φ are a non-normalized approximation of the autocorrelation and covariance respectively. Additionally, we have the following immediate result.

Proposition 2.12 For a stationary signal s and sufficiently large blocksize N, we find that

$$\varphi(\ell, r) \approx N \cdot \mathbb{E}\big(s[n-\ell]s[n-r]\big) = N \cdot \mathbb{E}\big(s[n]s[n-r+\ell]\big) \approx R(r-\ell).$$

2.2 An introduction to LPC

In essence, linear predictive coding is a modeling technique for any signal s with static spectral content. Namely, it states that such a static signal s can be seen as an excitation signal u —typically much smaller than s— shaped by some digital filter H. In other words,

$$S(z) = U(z)H(z) \quad \text{where } s \overset{\mathcal{Z}}{\to} S \text{ and } u \overset{\mathcal{Z}}{\to} U.$$

While s and u vary as the signal progresses and n becomes larger, the filter H remains fixed. It was mentioned that compression of signals using LPC always starts with partitioning s into *blocks* —called the *blocking* step— of some size N, and that is for precisely this reason. In this way, it becomes possible to more effectively model a signal with dynamic frequency content. When N is smaller, say one sixth of a second as is the standard, the assumption of stationary frequency content becomes easier to accept: over such a short period of time many types of signals are likely to be stationary, as is the case for the already mentioned speech formants, or a single note or chord in a song.

In practise, H is almost always chosen to be an *all-pole* AR filter, meaning that H has no zeros and is thus of the form

$$H(z) = \frac{G}{A\big({}^{1}\!/\!{}_{z}\big)} \qquad \text{for some polynomial } A\big({}^{1}\!/\!{}_{z}\big) = 1 + \sum_{k=1}^{p} a_k z^{-1} \text{ and } G \in \mathbb{R},$$

where G is a simple *gain* factor, although we will set $G = 1$ for our purposes. The main motivation for choosing an all-pole filter is that it will make certain computations, which we will encounter later, a lot easier. This boils down to the fact that we circumvent any recursive formulas, since we simply have

$$\begin{aligned}S(z) = U(z)H(z) &= U(z)\frac{1}{A\big({}^{1}\!/\!{}_{z}\big)} \\ S(z)A\big({}^{1}\!/\!{}_{z}\big) &= U(z).\end{aligned}$$

For which theorem 2.3 gives the time-domain equation

$$u[n] = s[n] + \sum_{j=1}^{p} a_j s[n-j]. \tag{1}$$

Moreover, this limitation will not have a large negative impact on the model's accuracy, considering that audio signals are almost always characterized by the *presence* of certain frequencies rather than their absence. Since zeros specifically model this absence, it intuitively makes sense to omit them in the filter.

With these choices set in stone, we can begin looking at methods for computing such a model. Before that however, we first state some terminology together with a few remarks.

Definition 2.13 We say a LPC all-pole filter $H(z) = 1/A(1/z)$ is of *order* p when A is of degree p.

Higher order filters allow for better spectral modelling but will require more computations. Too large filter orders will typically result in relatively little gain, so finding a suitable order will be a concern during computation.

Definition 2.14 (and remarks) Within LPC, the computation of $u[n]$ shown above is called the *analysis* step. Of course, $A(1/z)$ itself is a digital filter as well, so we summarize this step as

$$U(z) = S(z)A(1/z) \quad \text{such that} \quad u[n] = s[n] + \sum_{j=1}^{p} a_j s[n-j].$$

Note that this equation shows that given a filter $H(z) = 1/A\left(1/z\right)$ and a signal s, we can uniquely compute u.

Likewise, *synthesis* is the process of going back from the resulting u to s, which in turn is given by

$$S(z) = U(z)H(z) \quad \text{such that} \quad s[n] = u[n] - \sum_{j=1}^{p} a_j s[n-j].$$

Thus both analysis and synthesis are fixed given the coefficient set $\{a_j\}$ and block length N. The choices for computation we will make therefore concern only these parameters.

The resulting excitation signal $u[n]$ will be particularly relevant later, for computation, but for lossless compression especially.

Definition 2.15 The value $-\sum_{j=1}^{p} a_j s[n-j]$ is referred to as the *linear prediction* for $s[n]$ and its error is equal to $u[n]$. We will call this error the *residual* in n, and u itself the *residual signal.*

In the case of lossless compression, the signal s is encoded as the coefficient set $\{a_j\}$ together with all residuals $u[n]$, from which s can be uniquely reconstructed. When s is blocked, the first p values for s are also required as a *warm-up* for the synthesis step. In any case, computation of the coefficients a_j will seek to minimize the absolute values of $u[n]$ in some form. Afterwards, efficient compression of u itself will be essential as well, to which we will dedicate the last section, as well as chapter 3.

We will look at four approaches for computing the desired coefficient set. The standard Levinson-Durbin algorithm and the Burg algorithm using a lattice filter formulation will be discussed first, which both use the fixed blocksize formulation as mentioned before. This formulation, although often adequate in practise, leads to a discrepancy between the LPC model and original signal structure, where the block edges do not match up with the edges of the locally stationary sections present in most audio signals. We will revisit this notion with two new adaptive algorithm, which seeks to (efficiently) find blocksizes that match the audio signal in question.

2.3 The standard linear prediction algorithms

2.3.1 The Levison-Durbin algorithm

A least squares approach is standard for linear predictive coding, and it can be described directly from the linear prediction representation as in expression (1). A *realisation diagram* of such a system is a visual way of showing the computations, which is provided for this expression —the analysis step specifically— below. Realisations are not only effective ways of communicating a filter design, but are directly useable for hardware implementations as well. The realisation in figure 2.3 is generally referred to as the *transversal form* of the LPC filter.

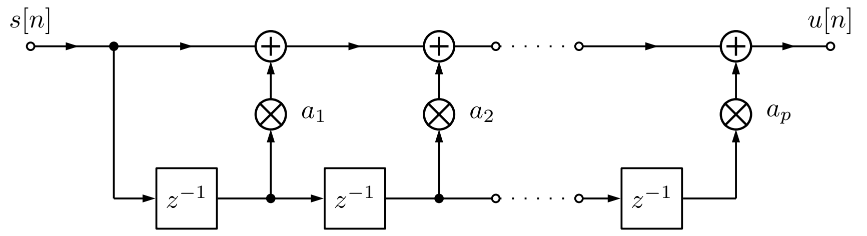


Figure 2.3: A realisation diagram of the synthesis function $u[n] = s[n] + \sum_{j=1}^{p} a_j s[n-j]$. Plus-nodes add two incoming values together, the z^{-1}-blocks mark the delay of the input signal by one —as is done by multiplication with z^{-1} in the $\mathcal{Z}$-domain— and $\otimes$-nodes indicate multiplication with the accompanying a_i.

For a signal s, LPC filter of order p and the corresponding residual signal u, the least squares approach seeks to minimize the *residual error* E, given by

$$E := \sum_{n=-\infty}^{\infty} u[n]^2 = \sum_{n=-\infty}^{\infty} \Big(s[n] + \sum_{j=1}^{p} a_j s[n-j]\Big)^2.$$

Since this expression is a second order polynomial in each of the p coefficients a_k, minimization can be done by setting their derivatives $\frac{\partial E}{\partial a_k}$ to zero, such that for $k \in \{1, 2, \cdots, p\}$ we have

$$0 = \frac{\partial E}{\partial a_k} = \sum_{n=-\infty}^{\infty} 2s[n-k]\Big(s[n] + \sum_{j=1}^{p} a_j s[n-j]\Big)$$

$$-\sum_{n=-\infty}^{\infty} s[n-k]s[n] = \sum_{n=-\infty}^{\infty}\sum_{j=1}^{p} a_j s[n-k]s[n-j]. \tag{2}$$

As discussed, in practise the partitioning of s in blocks is an integral part of the linear prediction procedure. Here we have two options, the *covariance method* and the more popular *autocorrelation method.*

Covariance method

If we assume s to equal zero outside the block range $\{0, 1, \cdots, N-1\}$, then expression

(2) reduces to

$$-\sum_{n=k}^{N-1} s[n-k]s[n] = \sum_{j=1}^{p} a_j \sum_{n=\max\{k,j\}}^{N-1} s[n-k]s[n-j].$$

With R the *autocorrelation vector* as found in definition 2.10 and φ the *covariance matrix* as found in definition 2.11 this equals $R(k) = \sum_{j=1}^{p} a_j \varphi(k,j)$ exactly. Putting all p expressions together we arrive at the linear system of

$$\begin{bmatrix} \varphi(1,1) & \cdots & \varphi(1,p) \\ \vdots & \ddots & \vdots \\ \varphi(p,1) & \cdots & \varphi(p,p) \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_p \end{bmatrix} = - \begin{bmatrix} R(1) \\ \vdots \\ R(p) \end{bmatrix},$$

which one can solve relatively efficiently using the symmetry of φ.

Autocorrelation method

If on the other hand we first rephrase our expression (2) with

$$\begin{aligned} \sum_{n=-\infty}^{\infty} \sum_{j=1}^{p} a_j s[n-k]s[n-j] &= \sum_{j=1}^{p} a_j \sum_{n=-\infty}^{\infty} s[n-k]s[n-j] \quad \text{(assuming convergence)} \\ &= \sum_{j=1}^{p} a_j \sum_{n=-\infty}^{\infty} s[n]s\big[n-|k-j|\,\big] \end{aligned}$$

and then apply the blocking step, we arrive at $-R(k) = \sum_{j=1}^{p} a_j R\big(|k-j|\big)$ and consequently the linear system of

$$\begin{bmatrix} R\big(|1-1|\big) & \cdots & R\big(|1-p|\big) \\ \vdots & \ddots & \vdots \\ R\big(|p-1|\big) & \cdots & R\big(|p-p|\big) \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_p \end{bmatrix} = - \begin{bmatrix} R(1) \\ \vdots \\ R(p) \end{bmatrix}.$$

Note that for stationary signals we have that $R\big(|k-j|\big) \approx \varphi(k,j)$ by proposition 2.12 and both methods become essentially equivalent. With this in mind, the autocorrelation method is the generally preferred technique because of the additional properties that the matrix $\big[R\big(|k-j|\big)\big]_{1\le k,j\le p}$ inhibits. Namely, that it is *Toeplitz* in addition to its symmetry.

Definition 2.16 We call a matrix $T \in \mathbb{R}^{p\times p}$ *Toeplitz* if all $2p-1$ diagonals are constant. Or more precisely, if for all i,j we have

$$T_{i,j} = T_{i+1,j+1}.$$

This characterisations follows immediately from the fact that $R\big(|(k+1)-(j+1)|\big) = R\big(|k-j|\big)$.

For Toeplitz matrices $T \in \mathbb{R}^{p\times p}$ the efficient Levinson-Durbin algorithm can be used to solve linear systems of the form $Ta = y$ for unknown a, which we will discuss here. It can solve this system with complexity $\mathcal{O}(p^2)$, a huge improvement over the $\mathcal{O}(p^3)$ necessary for linear systems with symmetric matrices. It is recursive over n, solving for $m = 1, 2, \ldots, p$ the linear system for the $m \times m$ upper-left submatrix $T^{(m)}$ of T, or equivalently —as a consequence of the Toeplitz characterisation— the $m \times m$ lower-right submatrix. Likewise, we write $y^{(m)} \in \mathbb{R}^m$ for the vector y up to and including the mth entry.

The main component of algorithm are the *forward vectors* $f^{(m)} \in \mathbb{R}^m$ and *backward vectors* $f^{(m)} \in \mathbb{R}^m$, which satisfy

$$T^{(m)} f^{(m)} = e_1^{(m)} \quad \text{and} \quad T^{(m)} b^{(m)} = e_m^{(m)},$$

with $e_i^{(m)}$ the i^{th} standard basic vector of dimension m.

Assuming that from the previous iteration we computed $f^{(m)}, b^{(m)}$ and $a^{(m)}$ such that this property holds together with $T^{(m)} a^{(m)} = y^{(m)}$, one iteration consists of the following steps. Note that for $m = 1$, finding $f^{(m)}, b^{(m)}$ and $a^{(m)}$ is trivial.

1. Calculate $f^{(m+1)}$ and $b^{(m+1)}$ by noting that, for some error terms $\varepsilon_f, \varepsilon_b \in \mathbb{R}$, we have

$$T^{(m+1)} \begin{bmatrix} f^{(m)} \\ \\ 0 \end{bmatrix} = \begin{bmatrix} e_1^{(m)} \\ \\ \varepsilon_f \end{bmatrix} \quad \text{and, since } T \text{ is Toeplitz,} \quad T^{(m+1)} \begin{bmatrix} 0 \\ \\ b^{(m)} \end{bmatrix} = \begin{bmatrix} \varepsilon_b \\ \\ e_m^{(m)} \end{bmatrix}.$$

   These two vectors differ from zero only at entry 1 and $m+1$, so for the right scalars α_f and β_f we find

$$T^{(m)} \left(\alpha_f \begin{bmatrix} f^{(m)} \\ \\ 0 \end{bmatrix} + \beta_f \begin{bmatrix} 0 \\ \\ b^{(m)} \end{bmatrix} \right) = e_1^{(m+1)}$$

   and likewise there exists scalars α_b and β_b such that this expression equals $e_{m+1}^{(m+1)}$. More precisely, by removing all redundant zero entries from the two vector equations, we find these scalars must satisfy

$$\begin{bmatrix} 1 & \varepsilon_b \\ \varepsilon_f & 1 \end{bmatrix} \begin{bmatrix} \alpha_f & \alpha_b \\ \beta_f & \beta_b \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \implies \begin{bmatrix} \alpha_f & \alpha_b \\ \beta_f & \beta_b \end{bmatrix} = \begin{bmatrix} 1 & \varepsilon_b \\ \varepsilon_f & 1 \end{bmatrix}^{-1} = \frac{1}{\varepsilon_f \varepsilon_b - 1} \begin{bmatrix} -1 & \varepsilon_b \\ \varepsilon_f & -1 \end{bmatrix}.$$

2. Calculate $a^{(m+1)}$ using the newly calculated backward vector by recognizing that

$$T^{(m+1)} \begin{bmatrix} a^{(m)} \\ \\ 0 \end{bmatrix} = \begin{bmatrix} y^{(m)} \\ \\ \varepsilon \end{bmatrix} \implies T^{(m+1)} \left(\begin{bmatrix} a^{(m)} \\ \\ 0 \end{bmatrix} + (y_{m+1} - \varepsilon) b^{(m+1)} \right) = y^{(m+1)}.$$

Note that for symmetric Toeplitz matrices, we can make the further simplification that for every m the forward and backward vectors are each others *reversals*, namely that $f_i^{(m)} = b_{m+1-i}^{(m)}$. For the autocorrelation method, it then follows that

$$\varepsilon_f = \sum_{i=1}^{m} R(m+1-i) f_i^{(m)} = \sum_{i=1}^{m} R(m+1-i) b_{m+1-i}^{(m)} = \sum_{i=1}^{m} R(i) b_i^{(m)} = \varepsilon_b,$$

such that explicit computation of b is never required. Writing $\overline{f^{(m)}} = b^{(m)}$ for the reversal of $f^{(m)}$, the Levinson-Durbin algorithm for LPC is therefore as follows.

Algorithm 2.17 (Levinson-Durbin (LPC version))

```
1: procedure LevinsonDurbinAlgorithm
2:     Calculate $R(i)$ for $i \in \{0, 1, \dots, p\}$
```

```
 3:     Set $f^{(1)} = \big[-1/R(0)\big]$ and $a^{(1)} = \big[-R(1)/R(0)\big]$
 4:     for $1 \leq m \leq p-1$ do
 5:         $\varepsilon_f \leftarrow \sum_{i=1}^{m} R(m+1-i) f_i^{(m)}$
 6:         $\varepsilon \leftarrow \sum_{i=1}^{m} R(m+1-i) a_i^{(m)}$
 7:         $f^{(m+1)} \leftarrow \frac{1}{\varepsilon_f \varepsilon_f - 1}\Big(-\big[f^{(m)} \;\; 0\big] + \varepsilon_f \big[0 \;\; \overline{f^{(m)}}\big]\Big)$
 8:         $a^{(m+1)} \leftarrow \big[a^{(m)} \;\; 0\big] - \big(R(m+1)+\varepsilon\big)\overline{f^{(m+1)}}$
 9:     end for
10: end procedure
```

Since the computation of $R(i)$ takes approximately pN operations, this algorithm has a complexity of $\mathcal{O}(pN+p^2)$, although since N is typically much larger than p, the computation of $R(i)$ dominates, such that the complexity becomes $\mathcal{O}(pN)$.

One popular extension is to expand the Toeplitz linear system to

$$\begin{bmatrix} R(0) & R(1) & \cdots & R(p) \\ R(1) & R\big(|1-1|\big) & \cdots & R\big(|1-p|\big) \\ \vdots & \vdots & \ddots & \vdots \\ R(p) & R\big(|p-1|\big) & \cdots & R\big(|p-p|\big) \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \vdots \\ a_p \end{bmatrix} = \begin{bmatrix} V \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

where we solely add the (unknown) variable V to the linear equations, whose value will result from the modified Levinson-Durbin recursion we provide below [5]. Writing $a^{(m)}$ for the coefficient vector including the additional 1, this reformulation allows for the unification of the forward vector $f^{(m)}$ and coefficient vector $a^{(m)}$, apart from a factor $V^{(m)} \in \mathbb{R}$ which satisfies $T^{(m)} a^{(m)} = V^{(m)} e_1^{(m)}$. Consequently,

$$\begin{aligned} a^{(m+1)} = V^{(m+1)} f^{(m+1)} &= \frac{V^{(m+1)}}{\varepsilon_f^2 - 1}\Big(-\big[f^{(m)} \;\; 0\big] + \varepsilon_f\big[0 \;\; \overline{f^{(m)}}\big]\Big) \\ &= \frac{V^{(m+1)}}{V^{(m)}\big(\varepsilon_f^2 - 1\big)}\Big(-\big[a^{(m)} \;\; 0\big] + \varepsilon_f\big[0 \;\; \overline{a^{(m)}}\big]\Big). \end{aligned}$$

Additionally, there is the extra requirement for $a_1^{(m+1)} = V^{(m+1)} f_1^{(m+1)} = 1$, which we respect by choosing $V^{(m+1)} = \big(1-\varepsilon_f^2\big)V^{(m)}$ such that

$$a^{(m+1)} = \big[a^{(m)} \;\; 0\big] - \varepsilon_f\big[0 \;\; \overline{a^{(m)}}\big].$$

And finally, $\varepsilon_f = \sum_{i=1}^{m} R(m+1-i) f_i^{(m)} = \sum_{i=1}^{m} R(m+1-i) a_i^{(m)}/V^{(m)}$, with which we can present the modified algorithm.

Algorithm 2.18 (Levinson-Durbin (Modified version))

```
1: procedure ModifiedLevinsonDurbinAlgorithm
2:     Calculate $R(i)$ for $i \in \{0, 1, \ldots, p\}$
3:     Set $a^{(1)} = \big[1\big]$ and $V^{(1)} = R(0)$
4:     for $1 \leq m \leq p$ do
5:         $\varepsilon_f \leftarrow \sum_{i=1}^{m} R(m+1-i) a_i^{(m)}/V^{(m)}$
6:         $V^{(m+1)} \leftarrow \big(1-\varepsilon_f^2\big)V^{(m)}$
7:         $a^{(m+1)} \leftarrow \big[a^{(m)} \;\; 0\big] - \varepsilon_f\big[0 \;\; \overline{a^{(m)}}\big]$
8:     end for
9: end procedure
```

This version of the algorithm is the one implemented by the FLAC reference encoder.

The Levinson-Durbin algorithm has some disadvantages, such as that it does not guarantee *numerical stability*. The notion of numerical stability is beyond the scope of this document —for more information we refer to [27]— although it should be noted that an alternative algorithm exists that does provide numerical stability which retains the p^2 complexity, namely the Bareiss algorithm for Toeplitz matrices [9]. Other disadvantages include that it is not easily possible to guarantee stability of the resulting LPC filter, and that when increasing the order p, the complete coefficient set needs to be altered. Nonetheless, the Levinson-Durbin is a straighforward and effecient algorithm.

We finish our discussing of the linear least squares approach with a short discussion on *windowing*. The blocking step introduces artifacts that negatively affect the effectiveness of the minimisation of E, namely it creates noise in the spectral content of the signal such that the found filter coefficients may not be completely accurate to the true spectrum. To combat this problem a *window* can be applied to the blocked signal, such that the signal magnitude at the edges of the block is significantly reduced, which mitigates these artifact at the cost of the blurring of the frequency content a bit. Many types of extensively studied windowing functions exist [22], one simple and prevalent example being the *Hamming window*.

Definition 2.19 For a blocksize of N, we define the *Hamming window* w as

$$w[n] = \begin{cases} \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N}\right)\right) & \text{for } 0 \leq n < N \\ 0 & \text{elsewhere.} \end{cases}$$

We can then replace the signal s with its windowed version $x[n] = s[n]w[n]$ during the blocking step, and apply algorithm 2.17 as normal. The use of windowing can make the linear prediction coefficients somewhat more accurate to the original signal, although it does not necessarily mean E becomes smaller if the computed coefficient set is applied to the non-windowed signal s. However, the windowing step is easily invertible, and LPC protocols may choose to include the windowing step as part of both the encoding and decoding algorithms to take full advantage of the advantages.

2.3.2 The lattice Burg algorithm

Besides the linear least squares methods, LPC can be considered in terms of a *lattice filter*. The standard lattice algorithm is the Burg algorithm, which we will derive and consequently build upon in this section and the next. Lattice filters have several advantages over the traditional approach described above, most notably their recursive nature and the existence of a simple characterisation of the stability of the resulting LPC filter. Moreover, computational efficiency is comparable to the least squares methods, provided some implementation details are accounted for.

Like the linear least squares methods, lattice filters are block-based, and we will once again refer to the blocksize as N. However, lattice filters will form the basis of the adaptive approach which we will present in the next session.

Definition 2.20 We define a *lattice filter* of order p for an input signal s by a set of *forward predictions* $f_m[n]$ and *backward predictions* $b_m[n]$ for $m \in \{0, 1, \ldots, p\}$, such that

$$f_0[n] = b_0[n] = s[n],$$

and recursively, for a set coefficients k_m with $m \in \{1, \ldots, p\}$,

$$\begin{aligned} f_m[n] &= f_{m-1}[n] + k_m b_{m-1}[n-1], \\ b_m[n] &= k_m f_{m-1}[n] + b_{m-1}[n-1]. \end{aligned}$$

We call $\{k_m\}$ the *reflection coefficients*. Lastly, we define the output signal u by $u[n] = f_p[n]$. Note that this $u[n]$ is once again a linear combination of $s[n-j]$ for $j \in \{0, 1, \dots, p\}$.

Just as for the standard filter model, we can visualise this filter in a realisation diagram.

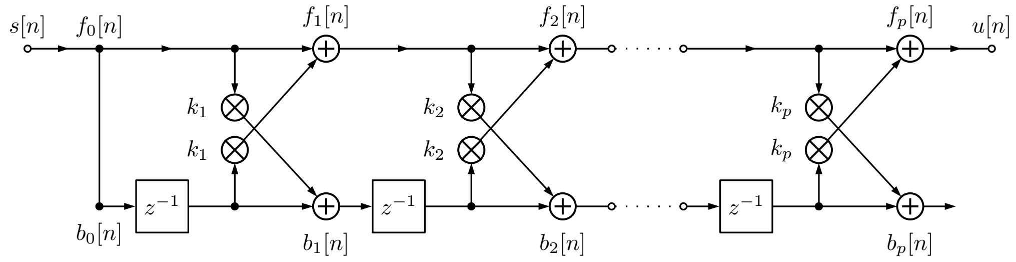


Figure 2.4: The realisation diagram of a lattice filter of order p. The predictions $f_m[n]$ and $b_m[n]$ have been added to visualise the recursive specification of the previous definition.

It is not immediately clear how the lattice filter and standard LPC filter relate to one another. We provide this relation below in theorem 2.22, but first we proof a relation between the forward and backward vectors of order m in the following lemma.

Lemma 2.21 If the forward prediction $f_m[n]$ is given by

$$f_m[n] = \sum_{j=0}^{m} q_k s[n-j], \quad \text{then} \quad b_m[n] = \sum_{j=0}^{m} q_k s[n-m+j].$$

Proof We provide a direct proof to provide some intuition about the lattice structure. A much shorter recursive proof is possible as well, but omitted here.

Looking at figure 2.4, we consider paths $\mathcal{P}$ through the diagram as follows: starting at $s[n]$ and for $\ell \in \{0, \dots, m-1\}$ we choose to walk along either the top or bottom of the diagram, at the ℓ^{th} forward or backward predictions respectively. As a final step for $j = m$, we walk along the top to arrive at $f_m[n]$.

Within all paths $\mathcal{P}$, we respect the computations we find along the way, thus walking along the bottom delays the input with 1, and changing lanes at step ℓ amounts to multiplication with k_ℓ. For the total delay in $\mathcal{P}$ we write $d_{\mathcal{P}}$, and for the product of all k_ℓ's the path encounters we write $k_{\mathcal{P}}$.

Note that $f_m[n]$ is then equal to the summation of the values along all 2^m possible paths in the filter:

$$f_m[n] = \sum_{j=0}^{m} q_k s[n-j] = \sum_{j=0}^{m} \sum_{\substack{\mathcal{P} \\ d_{\mathcal{P}}=j}} k_{\mathcal{P}} s[n-j].$$

Similarly however, this value would equal $b_m[n]$ if at step $j = m$ we would force all paths to go along the bottom instead. We define the complement $\overline{\mathcal{P}}$ of a path $\mathcal{P}$ simply as the path which chooses the other lane for every step $\ell \in \{0, \dots, m\}$. Note that whenever $\mathcal{P}$ goes along the bottom j times, $\overline{\mathcal{P}}$ goes along the bottom $m-j$ times. All lane changes remain the same, so only the delay is affected. The result follows:

$$\begin{aligned} b_m[n] &= \sum_{j=0}^{m} \sum_{\substack{\mathcal{P} \\ d_{\overline{\mathcal{P}}}=j}} k_{\overline{\mathcal{P}}} s[n-j] = \sum_{j=0}^{m} \sum_{\substack{\mathcal{P} \\ d_{\mathcal{P}}=j}} k_{\overline{\mathcal{P}}} s[n-m+j] \\ &= \sum_{j=0}^{m} \sum_{\substack{\mathcal{P} \\ d_{\mathcal{P}}=j}} k_{\mathcal{P}} s[n-m+j] = \sum_{j=0}^{m} q_k s[n-m+j]. \end{aligned} \qquad \square$$

We are now ready to describe how the reflection coefficients k_j relate to the LPC coefficients a_j.

Theorem 2.22 Given a signal s and a set of reflection coefficients $\{k_j\}_{1\le j\le p}$, we can explicitly define a set coefficients $\{a_j^{(p)}\}_{1\le j\le p}$ such that

$$u^{(p)}[n] = f_p[n] = s[n] + \sum_{j=1}^{p} a_j^{(p)} s[n-j].$$

More precisely, $a_j^{(p)}$ is defined recursively for $m \in \{0,\dots,p-1\}$ by

$$a_j^{(m+1)} = \begin{cases} k_{m+1} & \text{if } j = m+1, \\ a_j^{(m)} + k_{m+1} a_{m+1-j}^{(m)} & \text{for } 1 \le j < m+1. \end{cases}$$

Proof Using lemma 2.21 this follows quite quickly with induction on m. The base case for $m = 0$ is clear, since then $f_0[n] = s[n]$.

Now assume it holds for some m, thus that

$$u^{(m)}[n] = f_m[n] = s[n] + \sum_{j=1}^{m} a_j^{(m)} s[n-j]$$

$$\overset{\text{Lemma 2.21}}{\Longrightarrow} \qquad b_m[n] = \sum_{j=1}^{n} a_j^{(p)} s[n-m+j] + s[n-m].$$

We then find that

$$\begin{aligned} u^{(m+1)}[n] &= s[n] + \sum_{j=1}^{m+1} a_j^{(m+1)} s[n-j] \\ &= s[n] + \sum_{j=1}^{m} \big(a_j^{(m)} + k_{m+1} a_{m+1-j}^{(m)}\big) s[n-j] + k_{m+1} s[n-m-1] \\ &= s[n] + \sum_{j=1}^{m} a_j^{(m)} s[n-j] + k_{m+1} \Big(\sum_{j=1}^{m} a_{m+1-j}^{(m)} s[n-j] + s[n-m-1] \Big) \\ &= f_m[n] + k_{m+1} \Big(\sum_{i=1}^{m} a_i^{(m)} s[n-m-1+i] + s[n-m-1] \Big) \\ &= f_m[n] + k_{m+1} b_m[n-1] = f_{m+1}[n]. \end{aligned} \qquad \square$$

It was already mentioned that lattice filters have the advantage of the existence of an easy test for stability. The following result describes this characterisation.

Theorem 2.23 A lattice filter with reflection coefficients $\{k_j\}_{1\le j\le p}$ is stable if

$$|k_j| < 1 \quad \text{for all } j \in \{1, 2, \dots, p\}.$$

Proof We will use theorem 2.8, showing that all zeros ζ of the transfer function satisfy $\zeta < 1$.

The proof is based on Rouché's theorem. The theorem itself is beyond the scope of this document, so for the full statement and its proof we refer to [25]. Here we will use a direct consequence, namely that if two *holomorphic* functions f and g —of which the $\mathcal{Z}$-transformed functions are examples— satisfy $|f(z)| < |g(z)|$ on the unit circle $\{z \mid |z| = 1\}$, then f and $f + g$ have the same amount of zeros inside the unit disc $\{z \mid |z| < 1\}$, counted with multiplicity.

Now for the proof of our theorem. With induction on m, we take the $\mathcal{Z}$-transformation $A^{(m)}(1/z)$ of the coefficient set $\{a_j^{(m)}\}$, and using theorem 2.22 we find that

$$\begin{aligned} A^{(m)}\left(1/z\right) &= 1 + \sum_{j=1}^{m} a_j^{(m)} z^{-j} = 1 + \sum_{j=1}^{m-1} \left(a_j^{(m-1)} + k_m a_{m-j}^{(m-1)}\right) z^{-j} + k_m z^{-m} \\ &= A^{(m-1)}\left(1/z\right) + k_m z^{-m} A^{(m-1)}(z). \end{aligned}$$

Now consider

$$z^m A^{(m)}\left(1/z\right) = z^m A^{(m-1)}\left(1/z\right) + k_m A^{(m-1)}(z).$$

Then on the unit circle we have, if $k_m < 1$,

$$\left|z^m A^{(m-1)}\left(1/z\right)\right| = \left|A^{(m-1)}\left(1/z\right)\right| < \left|k_m A^{(m-1)}\left(1/z\right)\right| = \left|k_m A^{(m-1)}(z)\right|.$$

Rouché's theorem gives that $z^m A^{(m-1)}\left(1/z\right)$ has the same amount of zeros in the unit disc as $z^m A^{(m-1)}(1/z) + k_m A^{(m-1)}(z) = z^m A^{(m)}(1/z)$. Per the induction hypothesis, all $m - 1$ zeros of $A^{(m-1)}(1/z)$ lie within the unit disc, and multiplication with z^m necessarily adds to this a zero in 0. Thus the m zeros of $z^m A^{(m-1)}(1/z)$ lie within the unit disc, and it follows that all m zeros of $z^m A^{(m)}\left(1/z\right)$ —note that this is a polynomial of degree m— must lie in the unit disc as well. Consequently, so do all zeros ζ of $A^{(m)}\left(1/z\right)$, and they thus satisfy $\zeta < 1$. $\square$

Computation of the reflection coefficients is typically done step-by-step, as is the case for the Burg algorithm which we will describe below. Intuitively, the lattice structure lends itself well to a step-by-step procedure since the interchange between the backward and forward predictions allows newer reflection coefficients to affect the weight of earlier delayed samples in the system, something that is impossible for the standard transversal form. The combination with theorem 2.23 makes this particularly advantageous.

To begin we can decide we want to, for every $m \in \{1, \dots, p\}$, minimize the forward prediction f_m over all n. Because our goal is to make $u[n] = f_p[n]$ as small as possible, this is not an unreasonable place to start. Since $f_m[n]$ can be either negative or positive, we use the squared values as the error criterion.

Definition 2.24 We define the *forward squared error* F_m as

$$F_m := \sum_{n=m}^{N} f_m[n]^2 = \sum_{n=m}^{N} \left(f_{m-1}[n] + k_m b_{m-1}[n-1]\right)^2.$$

Note that we start the summation at m, since f_m is not defined for smaller n as a result of the backwards recursion. Then, we want to choose k_m such that F_m is minimized, which we will achieve by differentiating F_m with respect to k_m:

$$\begin{aligned}\frac{\mathrm{d}F_m}{\mathrm{d}k_m} &= \sum_{n=m}^{N} 2b_m[n-1]\big(f_{m-1}[n] + k_m b_m[n-1]\big)\\ &= 2\Big(\sum_{n=m}^{N} f_{m-1}[n]b_{m-1}[n-1]\Big) + k_m 2\Big(\sum_{n=m}^{N} b_{m-1}^2[n-1]\Big).\end{aligned}$$

For convenience going forward, we give the following definitions.

Definition 2.25 We define the *backward squared error* B_m and *centre squared error* C_m as respectively

$$B_m := \sum_{n=m}^{N} b_m[n-1]^2 \quad \text{and} \quad C_m := \sum_{n=m}^{N} f_m[n]b_m[n-1].$$

Since F_m is a second order polynomial in k_m and $F_m \geq 0$, setting $\mathrm{d}F_m/\,\mathrm{d}k_m$ to zero will give us our desired coefficient

$$k_m^F := -\frac{C_{m-1}}{B_{m-1}}.$$

On the other hand, as the backward predictions $b_{m-1}[n-1]$ are added to the forward predictions f_m every step of the recursion, it makes sense to try to minimize B_m as well. Repeating this procedure for B_m instead of F_m, we find the optimal coefficient

$$k_m^B := -\frac{C_{m-1}}{F_{m-1}}.$$

Note that since $F_{m-1}, B_{m-1} \geq 0$, both k_m^F and k_m^B have the same sign.

Intuitively, choosing a value between k_m^F and k_m^B should give good results. Moreover, the coefficient selection is motivated by theorem 2.23, in the sense of the following result.

Theorem 2.26 The *geometric mean* of k_m^F and k_m^B given by $k_m^G := \operatorname{sign}\big(k_m^F\big)\sqrt{k_m^F k_m^B}$ satisfies

$$\big|k_m^G\big| = \sqrt{k_m^F k_m^B} \leq 1.$$

Proof Considering the vectors $\vec{f} := \big(f_m[n]\big)_{m\leq n\leq N}$ and $\vec{b} := (b_m[n-1])_{m\leq n\leq N}$, we find by the Cauchy-Schwartz inequality that

$$\sqrt{k_m^F k_m^B} = \frac{\sqrt{C_{m-1}^2}}{\sqrt{F_{m-1}}\cdot\sqrt{B_{m-1}}} = \frac{\big|\langle\vec{f},\vec{b}\rangle\big|}{\|\vec{f}\|\cdot|\vec{b}\|} \leq 1. \qquad \square$$

Therefore, the value for k_m is chosen to be lower than or equal to k_m^G, such that the resulting filter is guaranteed to be stable. An often preferred choice is the *harmonic mean* of k_m^F and k_m^B, such that

$$k_m^H := \frac{2}{1/k_m^F + 1/k_m^B} = \frac{2C_{m-1}}{F_{m-1} + B_{m-1}}.$$

One important reason for this is that k_m^H can be, in contrary to the geometric mean, derived from a specific error criterion, namely the minimization of $F_m + B_m$. We omit the derivation here, and refer to [19] for further details.

Proposition 2.27 Since the harmonic mean is always smaller than the geometric mean for values of the same sign, we find that

$$\left|h_m^H\right| \leq \left|k_m^G\right| \leq 1. \quad \text{Moreover,} \quad \left|h_m^H\right| < \left|k_m^G\right| \leq 1 \quad \text{if } k_m^F \neq k_m^B.$$

This choice for k_m was first described by Burg [3], and it leads to the following algorithm.

Algorithm 2.28 (Burg)

```
1: procedure BurgsAlgorithm
2:     for $0 \leq m \leq p-1$ do
3:         $F_m \leftarrow \sum_{n=m}^{N} f_m^2[n]$
4:         $B_m \leftarrow \sum_{n=m}^{N} b_m^2[n-1]$
5:         $C_m \leftarrow \sum_{n=m}^{N} f_m[n] b_m[n-1]$
6:         $k_{m+1}^H \leftarrow 2C_m/(F_m + B_m)$
7:         for $m \leq n \leq N$ do
8:             $f_{m+1}[n] \leftarrow f_m[n] + k_{m+1}^H \cdot b_m[n-1]$
9:             $b_{m+1}[n] \leftarrow k_{m+1}^H \cdot f_m[n] + b_m[n-1]$
10:        end for
11:    end for
12:    $u \leftarrow f_p$
13: end procedure
```

From inspection one can tell that this algorithm has a complexity of $\mathcal{O}\left(pN\right)$, the same as the Levinson-Durbin approach described in the previous section. Although it should be noted that the amount of required computations is somewhat higher, since every value for p amounts to roughly 5 times as many computations. Nonetheless, the lattice method has clear advantages, and its formulation will be basis for the next section as well.

One such advantage is that F_m gives a measure of the current performance of the filter, since if m would be the final order of the lattice filter F_m would equal the residual squared error, in the same way that $F_p = \sum_{n=p}^{N-1} f_p[n]^2 = \sum_{n=p}^{N-1} u[n]^2$. As a consequence, one can decide on the optimal filter order —between 0 and p— by comparing all values for F_m.

2.4 Linear prediction algorithms with adaptive blocksizes

2.4.1 The adaptive covariance Burg algorithm

In this section we present a new algorithm based on the Burg algorithm stated in algorithm 2.28 which allows for sample-per-sample updating. This creates the possibility for the magnitude of u to be evaluated continuously, providing a method to determine which blocksize N is *optimal* for a given section of the signal s, all while maintaining a comparable computational complexity to the Burg algorithm.

It is motivated by a reexamination of the assumption on which the previous two method rely, where it is assumed that small enough blocksizes would make an audio signal sufficiently stationary for the all-pole filter to be an adequate approximation. In almost every case however, a fixed blocksize will ensure many stationary sections of s are unnecessarily cut into multiple parts, while each block by itself will likely contain a transition between two of these sections. The adaptive approach seeks to place the block edges on exactly these transitions.

As a preparation, we begin by reformulating the Burg algorithm in the following manner, which we lend from [19]. It is possible to calculate k_m^H without ever needing to explicitly compute the forward and backward vectors. Setting $a_0^{(m)} = 1$, theorem 2.22 gives that

$$F_m = \sum_{n=m}^{N} f_m^2[n] = \sum_{n=m}^{N} \Big(\sum_{j=0}^{m} \alpha_j^{(m)} s[n-j] \Big)^2.$$

Consequently, by taking the symmetric covariance matrix φ as found in definition 2.11, we find by the finiteness of all summations that

$$F_m \approx \sum_{\ell=0}^{m} \sum_{r=0}^{m} \alpha_\ell^{(m)} \alpha_r^{(m)} \varphi(\ell, r).$$

The approximation it due the difference in the summations to N. This difference is motivated by the desire to remove the dependence of m in the computation of φ, since then we only need to calculate it *once*, at the beginning of the algorithm. We set the summation to start earlier, namely at $\max\{\ell, r\}$. This can be interpreted as adding the first m “partial” forward predictions $f_0^2[0], f_1^2[1], \dots, f_{m-1}^2[m-1]$ to the mean squared error, which should not negatively impact the error criterion. Alternatively, the summation can be set to start at p.

Similarly, using lemma 2.21, we have

$$B_m = \sum_{n=m}^{N} \Big(\sum_{j=0}^{m} \alpha_j^{(m)} s[n-m-1+j] \Big)^2 \approx \sum_{\ell=0}^{m} \sum_{r=0}^{m} \alpha_\ell^{(m)} \alpha_r^{(m)} \varphi(m+1-\ell, m+1-r)$$

$$\text{and} \quad C_m \approx \sum_{\ell=0}^{m} \sum_{r=0}^{m} \alpha_\ell^{(m)} \alpha_r^{(m)} \varphi(\ell, m+1-r)$$

With this, we present the following modification of algorithm 2.28.

ALGORITHM 2.29 (COVARIANCEBURG)

```
1: procedure CovarianceBurg
2:     Calculate $\varphi$
3:     for $0 \le m \le p-1$ do
4:         $F_m \leftarrow \sum_{\ell=0}^{m} \sum_{r=0}^{m} \alpha_\ell^{(m)} \alpha_r^{(m)} \varphi(\ell, r)$
5:         $B_m \leftarrow \sum_{\ell=0}^{m} \sum_{r=0}^{m} \alpha_\ell^{(m)} \alpha_r^{(m)} \varphi(m+1-\ell, m+1-r)$
6:         $C_m \leftarrow \sum_{\ell=0}^{m} \sum_{r=0}^{m} \alpha_\ell^{(m)} \alpha_r^{(m)} \varphi(\ell, m+1-r)$
7:         $k_{m+1}^H \leftarrow 2C_m/(F_m + B_m)$
8:         Calculate $\{a_j^{(m+1)}\}$ from $k_{m+1}^H$ and $\{a_j^{(m)}\}$
9:     end for
10:    $u \leftarrow f_p$
11: end procedure
```

This algorithm has a complexity of $\mathcal{O}(pN + p^3)$, however it is dominated by the calculation of the covariance matrix φ —with complexity $\mathcal{O}(pN)$— as N is typically much larger than p^2, which once again gives us an algorithm in $\mathcal{O}(pN)$. Note that this complexity for the calculation of φ is due to the fact that one can calculate φ from the autocorrelation vector R —in turn computable with complexity $\mathcal{O}(pN)$– with just $\mathcal{O}(p^2)$ additional operations, as we described at the end of the first section of this chapter. The calculation of F_m, B_m and C_M can be sped up significantly, which is described in [19]. For our purposes however this version already enables us to formulate the adaptive approach, and the methods described in this paper can be applied afterwards as well.

A central observation is that the covariance matrix φ can be calculated sample-per-sample, and as a consequence, we can give immediate estimates for the reflection coefficients $\{k_m\}$ based on $\varphi^{(n)}$, where

$$\varphi^{(n)}(\ell, r) = \sum_{i=\max\{\ell,r\}}^{n} s[i-\ell]s[i-r].$$

Since the values for $\{k_m\}$ are calculated step-by-step we can compute the next k_m every sample without the updated covariance measure affecting the results negatively. Most importantly, we can compute k_m relatively cheaply as we found in the covariance Burg algorithm. Next, since we seek to find the blocksize for which the prediction is optimal, we are in need of some continuous measure for the magnitude of the residuals on which this optimality can be based. Fortunately, the p^{th} average forward squared error $\frac{1}{N-p}F_p$ fits this description perfectly as $\sum_{n=p}^{N-1} f_p[n] = \sum_{n=p}^{N-1} u[n]$. To this end, we choose to iteratively recalculate the coefficient set by setting $m = n \bmod p$, an idea which we borrow from [14].

This leads to algorithm 2.30 below. Note that this algorithm increases the complexity to $\mathcal{O}\left(p^2N\right)$, although we will provide an alternative formulation with complexity $\mathcal{O}\left(pN\right)$ later. To allow for the covariance approximation to "warm up" we start with a *minimal blocksize* $N_{\min}$, which can be as small as ±200 samples, but it can be set to 0 as well. Additionally, it is possible to specify a maximum blocksize $N_{\max}$ in a self-evident manner, although we omit it here.

Algorithm 2.30 (AdaptiveBurg)

```
1: procedure AdaptiveBurg
2:     Initialise $\varphi$ for $0 \le n < N_{\min}$
3:     for $n \ge N_{\min}$ do
4:         $m \leftarrow (n - N_{\min}) \bmod p$
5:         Update $\varphi$ with $s[n]$
6:         $F_m \leftarrow \sum_{\ell=0}^{m} \sum_{r=0}^{m} \alpha_\ell^{(m)} \alpha_r^{(m)} \varphi(\ell, r)$
7:         $B_m \leftarrow \sum_{\ell=0}^{m} \sum_{r=0}^{m} \alpha_\ell^{(m)} \alpha_r^{(m)} \varphi(m+1-\ell, m+1-r)$
8:         $C_m \leftarrow \sum_{\ell=0}^{m} \sum_{r=0}^{m} \alpha_\ell^{(m)} \alpha_r^{(m)} \varphi(\ell, m+1-r)$
9:         $k_{m+1}^H \leftarrow 2C_m/(F_m + B_m)$
10:        Calculate $\left\{a_j^{(m+1)}\right\}$ based on $k_{m+1}^H$ and $\left\{a_j^{(m)}\right\}$
11:        if $m = p-1$ and $\frac{1}{n-p}F_{m+1}$ ($= \frac{1}{n-p}F_p$) is minimal then
12:            $N \leftarrow n$
13:            break
14:        end if
15:    end for
16:    $u \leftarrow f_p$
17: end procedure
```

Clarification of the statement "$\frac{1}{n-p}F_p$ is *minimal*" is in order. It should be interpreted in the sense that at the n for which $F_p/(n-p)$ is minimal, we have reached the end of a stationary section of the signal, and increasing the blocksize further suddenly results in *greater* values for $F_p/(n-p)$, whereas before it was decreasing instead. For determining whether $F_p/(n-p)$ is minimal multiple approaches are possible, and we highlight one method which we found to be effective here. The procedure above is a simplification, as it is not possible to decide whether $F_p/(n-p)$ is minimal without knowing its subsequent values. The algorithm we will discuss will therefore replace this if-statement in line 11-14 of algorithm 2.30.

The first thing to note is that we are trying to find a minimum of a function *while* we are evaluating it. There is therefore a balance between accuracy and execution time, in the sense that one can choose to evaluate the function further to increase the certainty of the minimisation, but this will be at the cost of computational efficiency. Luckily, for stationary signals we can make the assumption that the first local minimum will in fact also be the global minimum, since once the stationarity is broken the linear predictor will need to average over two incompatible —in terms of stationarity— sections of the signal, in which it will perform worse.

Moreover, the values for F_p lead to a somewhat noisy function depending on the type of audio analysed. In general, signals that lend itself better to linear prediction result in smoother graphs and subsequently better minimization, while signals that do not, such as a noisy signal for instance, tend to make minimization harder. In any case, this further complicates the previous point, as the "first local minimum" we are after will not truly be the first, and steps have to be taken to ensure the correct minimum is found.

Lastly, there is an important balance between the values of $F_p/(n-p)$ and their corresponding blocksizes. Whenever a value of $F_p/(n-p)$ is above but very close to the absolute minimum previously found, while the corresponding blocksize is much larger, it will be favored over the absolute minimum. This is because even when this new $F_p/(n-p)$ represents a blocksize such that the signal is stationary, noise can interfere and cause the error to be a bit higher.

All this being said, the main idea of the minimization is simple. We keep track of the current absolute minimum F_{best}, together with its corresponding blocksize N_{best} and covariance matrix $\varphi^{(N_{\text{best}})}$, and whenever we find a $F_p/(n-p)$ that is lower we set $F_{\text{best}} \leftarrow F_p/(n-p)$ and update N_{best} and $\varphi^{(N_{\text{best}})}$. When no improvement has been made for N_{search} samples, we set the final blocksize to equal N_{best}, revert to the value $\varphi^{(N_{\text{best}})}$ and recalculate the reflection coefficients.

As said, larger blocksizes are preferred when errors are close, which is why when $F_p/(n-p) \in (F_{\text{best}}, r \cdot F_{\text{best}}]$ for a fixed $r \in \mathbb{R}$ that denotes the *error range*, we do update N_{best} and $\varphi^{(N_{\text{best}})}$ but leave F_{best} unchanged. The following procedure gets called every time a new F_p is calculated in algorithm 2.30, thus every p samples. In practise, we found $N_{\text{counter}} = 2048$ and $r = 1.005$ to be effective parameters of this detection algorithm.

ALGORITHM 2.31 (DETECTION OF MINIMALITY)

```
1:  procedure DetectMinimal
2:      Set $r$ and $N_{\text{counter}}$
3:      counter $\leftarrow 0$
4:      if $F_p/(n-p) \leq F_{\text{best}}$ then
5:          $F_{\text{best}} \leftarrow F_p/(n-p)$
6:          $N_{\text{best}} \leftarrow n+1$
7:          $\varphi^{(N_{\text{best}})} \leftarrow \varphi^{(n)}$
8:          counter $\leftarrow 0$
9:      else if $F_p/(n-p) \leq r \cdot F_{\text{best}}$ then
10:         $N_{\text{best}} \leftarrow n+1$
11:         $\varphi^{(N_{\text{best}})} \leftarrow \varphi^{(n)}$
12:         counter $\leftarrow 0$
13:     else
14:         counter $\leftarrow$ counter $+ p$
15:         if counter $\geq N_{\text{counter}}$ then
16:             $N \leftarrow N_{\text{best}}$
```

```
17:             Calculate $\{k_j\}$ using $\varphi^{(N_\text{best})}$ and algorithm 2.29
18:             **break**
19:         **end if**
20:     **end if**
21: **end procedure**
```

Since during most of the analysis F_p is decreasing, it should be noted that updating $\varphi^{(N_\text{best})}$ can take significant time if done naively, but an $\mathcal{O}(1)$ implementation is possible if it is combined with the per-sample updating of φ.

2.4.2 A modification of the adaptive algorithm

As mentioned, the complexity of the adaptive algorithm is $\mathcal{O}(p^2N)$, which is somewhat worse than the $\mathcal{O}(pN)$ of the Levinson-Durbin and covariance Burg algorithms. In this section we provide a modified version of the algorithm which has complexity $\mathcal{O}(pN)$, although possibly at the loss of some of the blocksize detection accuracy.

To begin, we found that in most cases the blocksize detection requires only small filter orders to be effective, which one can motivate intuitively by the fact that even small orders will capture the fundamental frequencies in stationary sections of the signal. Setting p to be smaller and constant during the iteration, say to $p' = 3$ or $p' = 4$, while updating φ with the full order p, detection of the optimal N can be done efficiently, where after the N_best has been found the full order can be computed in the relatively small complexity of $\mathcal{O}\left(p^3\right)$ operations using the already calculated φ. With this modification, the calculation of F_m, B_m and C_m can be done in $(p')^2 \in \mathcal{O}(1)$ operations.

Sadly the updating of φ still requires p^2 operations which preserves the $\mathcal{O}(p^2N)$ complexity. This is where the modification comes in. As is done with the Levinson-Durbin approach, we can estimate $\varphi(\ell, r)$ with the autocorrelation matrix $R\big(|\ell - r|\big)$, which can be updated in just p operations. When the best blocksize is found, $\varphi^{(N_\text{best})}$ can be calculated from R_best, which replaces $\varphi^{(N_\text{best})}$ in algorithm 2.31. This results in an algorithm of complexity $\mathcal{O}(pN)$, which we will name the autocorrelation adaptive Burg algorithm.

2.4.3 Evaluation of the blocksize detection

We will end this section with a short discussion on the blocksize detection of the two adaptive algorithms. Discussing this formally is difficult, since we would require human-annotated audio data —where the “block edges” would be provided by a human listener— to compare against the block edges the algorithms found. This specific need is not readily available, so we will only provide a informal discussion here. Namely, accuracy of the blocksize detection was evaluated by looking at the *error graphs* of F_p/n in every block, as well as by adding soft pops at the block edges in the audio data and listening if they matched up with the music.

For this evaluation, we will be using excerpts from the music tracks defined in table 2.1 which we specify in the next section. To begin, we show two of these error graphs for the adaptive covariance Burg algorithm. One for a stationary block where detection goes as expected, and one for a non-stationary block where detection becomes more difficult.

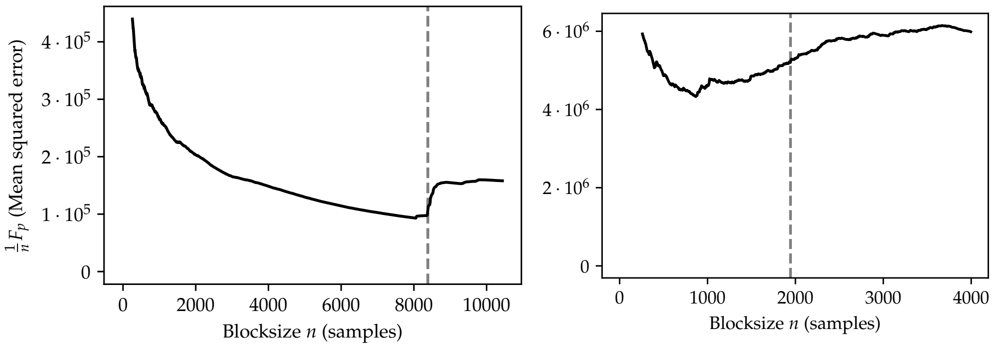


Figure 2.5: Left: Example of an error graph and block detection for a stationary block. Right: Example of an error graph for a non-stationary / noisy block. Both use the covariance adaptive Burg algorithm with detection parameters $N_{\text{range}} = 2048$ and $r = 1.01$. The dotted line denotes the final chosen blocksize.

Depending on the song in question —once again chosen from table 2.1— we find different results here. All show the left pattern of figure 2.5 often, especially the jazz and electronic tracks, and conversely the rock and noisy recording show more of the right pattern, although both still occur.

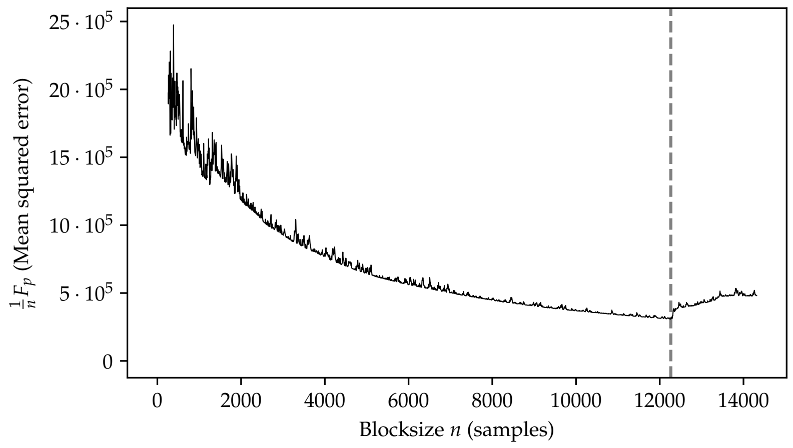


Figure 2.6: Example of an error graph for the block detection, using the autocorrelation adaptive Burg algorithm with detection parameters $N_{\text{range}} = 2048$ and $r = 1.01$. The dotted line denotes the final chosen blocksize.

The adaptive autocorrelation Burg algorithm shows these same patterns, however the error estimation is much less accurate, which can be seen in figure 2.6 above. The general shape is maintained, although one can imagine this noise may have negative effects on the blocksize detection in some scenarios.

Lastly, for both methods, it can occur whenever fragments with less pronounced transitions are analysed, practically monotonous decreasing graph are generated, such that the algorithm chooses the maximum blocksize as the optimal one, even though the audio data suggests otherwise.

Now for part two of our discussion, where we listen to the pops at the block edges and try to see if they match up with the audio content of the song. The standard adaptive covariance Burg algorithm performs well in this regard for most audio tracks. The pops are rhythmic and follow along with the chords, however some chord changes are ignored and sometimes too many changes are picked up. We also note that blocksizes tend to be much larger than the standard 4096 samples of the fixed-blocksize algorithms, a result from the accordance with the audio. Whenever the audio gets noisier, as is the case for the noisy audio recording and the rock track, the detection suffers. The pops become more random, and match up with the audio only sometimes.

The autocorrelation modification in general makes the detection worse, sometimes adding more unnecessary pops creating a lot of very small blocks in short bursts, and sometimes not detecting any blocks at all until the maximum blocksize is reached. Although in the cases that the detection worked well for its covariance counterpart, the autocorrelation performed reasonably well too. Considering the significant computational improvements this modification presents —for more information about this, see chapter 4— this method could however still be the preferred.

Overall, even though the block size detection is definitely not perfect, both methods perform well in general, and they can be especially reliable for audio with clear borders between the different stationary sections. The algorithms can afford much larger blocksizes while not negatively impacting the average magnitude of the residuals in the block. In chapter 4, we will study if this translates in better compression ratios, providing an experimental comparison with the standard fixed-blocksize Levinson-Durbin and covariance Burg algorithms. In general, the blocksize detection shows promise, and further expansions could seek to improve the border recognition when the borders in the audio signal are vague.

2.5 The distribution of the residuals

In the previous section we have discussed the minimization of the residuals $u[n]$ with the 2-norm as our error measure. For lossless compression, efficient encoding of these relatively small residuals is of importance, and knowing the patterns they inhibit will be essential in choosing and designing suitable encoders for this task. In this section we will show that residuals tend to behave more conform to the *Laplacian distribution*, instead of the normal distribution one would typical expect. The Laplacian distribution has a close connection with the discrete-valued *2-sided geometric distribution*, which we will use in our analysis of Golomb-Rice codes in chapter 3. In this chapter we give the definitions of the two distributions as well. We will provide comparisons between the four algorithms discussed, namely for the modified Levinson-Durbin algorithm (algorithm 2.18), the covariance Burg algorithm (algorithm 2.29), as well as our proposed adaptive Burg algorithm (algorithm 2.30) and autocorrelation adaptive Burg algorithm (subsection 2.4.2).

The comparison uses a limited selection of five music tracks, although since every song will be made up of around 1000 blocks, this should provide statistically significant results. Music was selected to vary in genre and instrumentation.

Song title	**Artist**	**Genre**	**Length**
Promise of the World (Instrumental)	Joe Hisaishi	Orchestral	4:10
Dream Lantern	RADWIMPS	Rock	2:12
Hello	-	Electronic	2.04
The Frog Prince	Aivi Tran	Jazz	3:47
Ballade no. 1 (Noisy recording)	Frédéric Chopin	Classical (piano)	11:09

Table 2.1: The audio tracks used for the comparison.

We begin with a visual indication of the distribution of the residuals, and how well either the normal or Laplacian distribution fits. Fitting of the two distributions is done by way of maximum likelihood estimation, where additionally the mean is assumed to be 0. Normal distribution uses the mean squared value (the standard deviation), and the Laplacian distribution uses the mean absolute value. Another estimation method, based on the results from chapter 3, is included later in this section. This method is based on the estimation of the parameter p for the 2-sided geometric distribution, with which the corresponding Laplacian parameter $-1/\ln(1-p)$ can be calculated. For the adaptive algorithms, we set the detection parameters as $N_{\text{range}} = 2048$ and $r = 1.01$.

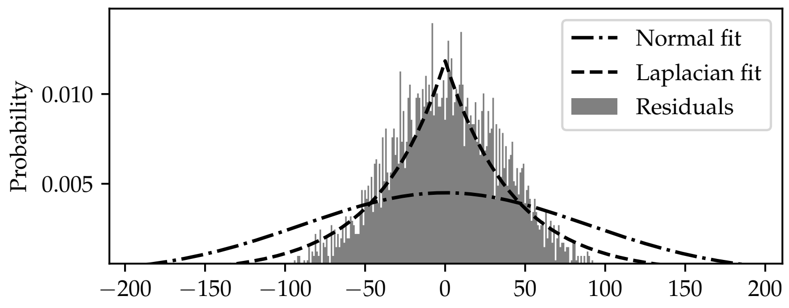


Figure 2.7: Residuals of a single block (one channel) computed with the fixed-blocksize Levinson-Durbin algorithm, blocksize 4096 and LPC order 10, together with the normal and Laplacian MLE fits. Non-noisy audio, block 81 from the orchestral “Promise of the World (Instrumental)” from table 2.1.

This graph indicates the Laplacian distribution might be a better fit in most scenarios. We contrast this with the following graph, which shows that this might not always be the case. The song used here is the noisy recording as mentioned in table 2.1, although this pattern shows up for in other tracks as well, especially when harmonic content tends to be less pronounced.

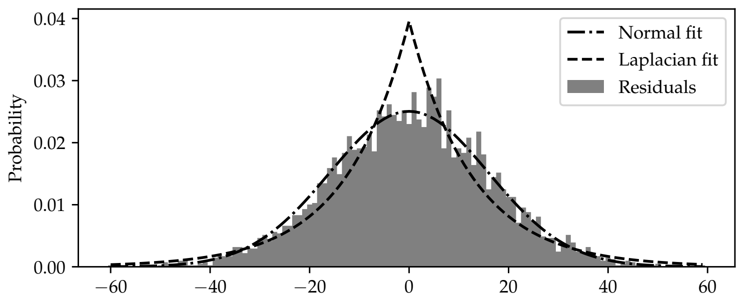


Figure 2.8: Residuals of a single block (one channel) computed with the fixed-blocksize Levinson-Durbin algorithm, blocksize 4096 and LPC order 10, together with the normal and Laplacian MLE fits. Noisy audio, block 62 from the classical “Ballade no. 1 (Noisy recording)” from table 2.1.

Because of this, it is unclear a priori which distribution should be preferred. We provide the formal comparison in the form of the average mean squared error between the residuals and the two distributions for different LPC orders. In this comparison we exclude blocks where the residuals are zero almost everywhere, in which case (continuous) estimators are less applicable and lead to inaccurate results. We find that the Laplacian distribution does in fact give better results, which we show in the following table.

LPC order 5:

Algorithm	**Normal distribution**	**Laplacian distribution (Max. likelihood)**	**Laplacian distribution (Geometric)**
Levinson-Durbin Fixed	$1.3621 \cdot 10^{-04}$	$2.3660 \cdot 10^{-05}$	$1.4351 \cdot 10^{-05}$
Covariance Burg Fixed	$1.5623 \cdot 10^{-04}$	$2.5388 \cdot 10^{-05}$	$1.5438 \cdot 10^{-05}$
Covariance Burg Adaptive	$6.6789 \cdot 10^{-05}$	$1.5611 \cdot 10^{-05}$	$9.2521 \cdot 10^{-06}$
Autocorrelation Burg Adaptive	$1.0657 \cdot 10^{-04}$	$1.6531 \cdot 10^{-05}$	$1.0540 \cdot 10^{-05}$

LPC order 10:

Algorithm	**Normal distribution**	**Laplacian distribution (Max. likelihood)**	**Laplacian distribution (Geometric)**
Levinson-Durbin Fixed	$1.4171 \cdot 10^{-04}$	$2.0364 \cdot 10^{-05}$	$1.2780 \cdot 10^{-05}$
Covariance Burg Fixed	$1.6516 \cdot 10^{-04}$	$2.2798 \cdot 10^{-05}$	$1.4194 \cdot 10^{-05}$
Covariance Burg Adaptive	$7.0655 \cdot 10^{-05}$	$1.4124 \cdot 10^{-05}$	$8.4627 \cdot 10^{-06}$
Autocorrelation Burg Adaptive	$1.1781 \cdot 10^{-04}$	$1.6039 \cdot 10^{-05}$	$1.0555 \cdot 10^{-05}$

LPC order 15:

Algorithm	**Normal distribution**	**Laplacian distribution (Max. likelihood)**	**Laplacian distribution (Geometric)**
Levinson-Durbin Fixed	$1.4480 \cdot 10^{-04}$	$2.1344 \cdot 10^{-05}$	$1.3182 \cdot 10^{-05}$
Covariance Burg Fixed	$1.6924 \cdot 10^{-04}$	$2.3772 \cdot 10^{-05}$	$1.4606 \cdot 10^{-05}$
Covariance Burg Adaptive	$7.1844 \cdot 10^{-05}$	$1.4439 \cdot 10^{-05}$	$8.6394 \cdot 10^{-06}$
Autocorrelation Burg Adaptive	$1.2138 \cdot 10^{-04}$	$1.6829 \cdot 10^{-05}$	$1.0986 \cdot 10^{-05}$

Table 2.2: The mean squared error of the residuals with a fitted distribution, for the normal distribution and the Laplacian distribution estimated with the maximum likelihood estimator and the aforementioned estimation based on the 2-sided geometric distribution. The results are averaged over the audio tracks from table 2.1. Each algorithm used a fixed LPC order of 5, 10 or 15.

Although the results are quite close for the two distributions, the Laplacian fit performs better for every algorithm consistently across LPC orders, especially for the estimation based on the 2-sided geometric distribution which improves on the maximum likelihood estimator in every case. Even if the normal distribution might be a more suitable description in some scenarios, on we can conclude that the Laplacian fit is more accurate on average. Furthermore, the adaptive algorithms show lower errors for all three categories, although since this improvement is small this can be attributed to the larger blocksizes which results form these algorithms.

3 Golomb codes

Golomb codes are an important class of entropy coders, and they will be central for the efficient encoding of the residuals provided from the LPC step, although this chapter will in principle be independent from any knowledge of LPC. Entropy coders utilize the known or estimated probability distribution the given data satisfies in order to represent it in an optimal manner. They are named such because they typically satisfy an important criteria for optimality, namely that the encoded bits-per-sample is close to the *Shannon entropy* of the distribution. We will clarify this statement, for which we need some mathematical background, in the next section. Golomb codes describe a class of operators which are suitable for compression of data which is, in some form, exponentially distributed.

The standard compression technique for data sets with known probability are the more well-known Huffman codes. One disadvantage in our case is the fact that these codes, at least in their basic form, do not allow for encoding of arbitrarily sized numbers, which the Golomb codes support by construction. Moreover, the prominent Golomb-Rice codes can be implemented efficiently using computer arithmetic, and do not require large codebooks associated with many Huffman codes. For this reason, we these Golomb-Rice codes will receive particular attention throughout this chapter.

Golomb-Rice codes are used by FLAC for the entropy encoding of the LPC residuals, as is the case for many different lossless formats. Other methods include *arithmetic coding*, or the more recent *asymmetric numeral systems*, although we will not dedicate much time to them here. MPEG-4 ALS for instance, utilizes a combination of a coding scheme closely related to efficient arithmetic codes for the bulk of the residuals, while still relying on the Golomb-Rice codes for the largest outliers [23].

We will begin with some mathematical background which both places Golomb codes in broader context of entropy coders, as well as provide some central theorems on which the latter halve of this chapter is based. However, the practical sides of these codes do not require a lot of mathematical background, and as a result section 3.2 can be understood and applied independent of any other section given in this chapter. After their introduction, we provide a demonstration and (crude) proof of the optimality of the Golomb-Rice codes for 2-sided geometrically distributed data —which the LPC residuals follow— specifically.

3.1 Mathematical preliminaries

This section will provide a short introduction to entropy codes, and we will state some central results, although proofs will be omitted. To start, for a given symbol set $\mathcal{A}$, we define an *entropy code* $\mathcal{C}$ as a map from $\mathcal{A}^+$ to a *codeword* in $\{0,1\}^+$, with $X^+ := \{x_1x_2\cdots x_n \mid x_i \in X, n \geq 1\}$. When every symbol $x \in \mathcal{A}$ is encoded one at the time, such that

$$\mathcal{C}(x_1x_2x_3\cdots) = \mathcal{C}(x_1)\mathcal{C}(x_2)\mathcal{C}(x_3)\cdots$$

we call $\mathcal{C}$ a *symbol code.* Note that these codes can be unambiguously defined as a map $\mathcal{A} \to \{0,1\}^+$.

An important property entropy coders will be required to abide by for lossless compression specifically is the notion of unique decodability.

Definition 3.1 We call an entropy code $\mathcal{C}$ *uniquely decodable* if for $w, w' \in \mathcal{A}^+$,

$$\mathcal{C}(w) = \mathcal{C}(w') \quad \Longrightarrow \quad w = w'.$$

Or in other words, such that $\mathcal{C}$ is injective.

Informally, this means that given a codeword $\mathcal{C}(w)$, one will be able to reconstruct w if $\mathcal{C}$ is known, thus that a well-defined *decoder* $\mathcal{D}\colon \{0,1\}^+ \to \mathcal{A}^+$ exists such that $\mathcal{D} \circ \mathcal{C} = \mathrm{id}_{\mathcal{A}^+}$. As a result, symbol transmission via $\mathcal{C}$ will be lossless.

An import class of uniquely decodable symbol codes are the prefix codes.

Definition 3.2 A symbol code $\mathcal{C}$ is a *prefix code* if for any $w \in \mathcal{A}^+$ no $w' \in \mathcal{A}^+$ exists with $\mathcal{C}(w)a = \mathcal{C}(w')$ for any $a \in \{0,1\}^+$, such that $\mathcal{C}(w)$ is not a prefix of any other codeword $\mathcal{C}(w')$.

Golomb codes, as is the case for most entropy codes including Huffman and arithmetic codes, are examples of prefix codes.

Intuitively, it is clear that all prefix codes are uniquely decodable: one can iterate through a codeword $\mathcal{C}(x_1)\mathcal{C}(x_2)\mathcal{C}(x_3)\cdots$ from left to right until detecting any $\mathcal{C}(x_i)$ and, by the prefix characterisation, be guaranteed that it is not part of another codeword. Moreover, it can be shown that prefix requirement imposes no further restrictions on the effectiveness of codes than already imposed by unique decodability for symbol codes. We will omit the technicalities here, and refer to [6] for details.

For compression, the goal of an entropy coder is to minimize the codeword lengths associated with the input alphabet. To this end, we consider a random variable $X \sim P$ on $\mathcal{A}$ for some distribution P and study $\mathcal{C}(X)$. Writing $\left|\mathcal{C}(w)\right|$ for the amount of bits in a codeword $\mathcal{C}(w)$, our goal becomes to minimize the expected codeword length $\mathbb{E}\big(|\mathcal{C}(X)|\big)$.

Central in this assessment of the effectiveness of entropy codes is the notion of entropy –hence their name— which both provides a measure for the performance of codes and states the limits of lossless compression. Entropy is determined by the symbol set and the probability distribution thereon. As is common in information theory, we will write ‘log’ for the binary logarithm of base 2, while the natural logarithm will be denoted with ‘ln’.

Definition 3.3 (Shannon entropy) For a random variable $X \sim P$ with $X \in \mathcal{A} = \{x_1, x_2, \ldots\}$ and probability distribution P on $\mathcal{A}$, we define the *Shannon entropy* of X, and its distribution P, as

$$H(X) = H(P) \coloneqq \sum_{x \in \mathcal{A}} \mathbb{P}(X = x) \log \frac{1}{\mathbb{P}(X = x)}.$$

The Shannon entropy is an evaluation of the *information content* of any random variable X, and the bits required to encode such information. Consequently, $H(X)$ provides a lower bound on the codeword lengths for lossless codes, in the sense of the following fundamental theorem.

Theorem 3.4 Let $\mathcal{C}\colon \mathcal{A} \to \{0,1\}^+$ be an uniquely decodable code and $X \sim P$ a random variable on $\mathcal{A}$, then

$$H(X) \leq \mathbb{E}\Big(\big|\mathcal{C}(X)\big|\Big).$$

Optimality of symbol codes therefore comes down to showing the expected codeword is close to the entropy of the symbol distribution. In practise, demonstrating that $\mathbb{E}\big(|\mathcal{C}(X)|\big) \in \big[H(X), H(X)+1\big]$ is typical. In particular, for any random variable X with $\mathbb{P}(X = x) \neq 0$ for only a finite amount of symbols, the efficient Huffman's coding algorithm can be used to construct an optimal symbol code $\mathcal{C}_H$ which satisfies $\mathbb{E}\big(|\mathcal{C}_H(X)|\big) \in \big[H(X), H(X)+1\big]$ [6]. In many cases, further enhancements can be made when straining away from symbol codes, with the family of *arithmetic codes* —and its more recent computational improvement in the form of *asymmetric numeral systems*— being the most notable example [8].

In the other case, when $\mathbb{P}(X = x) \neq 0$ for non-finite x's in $\mathcal{A}$, these algorithms can generally not be applied. However, distributions of this form can always be used to order $\mathcal{A}$ as $\mathcal{A} = \{x_1, x_2, x_3, \cdots\}$ such that the probability is *monotonously decreasing*, namely that $\mathbb{P}(X = x_i) \geq \mathbb{P}(X = x_{i+1})$. This gives rise to the so-called *universal codes*, which we define as follows.

Definition 3.5 (Universal codes) We call a prefix code $\mathcal{C} \colon \mathcal{A} \to \{0,1\}^+$ *weakly universal* when it can be applied to $\mathcal{A} = \mathbb{N}$, such that they can be used to encode alphabets of arbitrary size.

Additionally, $\mathcal{C}$ is called *universal* if for any monotonously decreasing distribution P and for $X \sim P$, we have that $\mathbb{E}\big(|\mathcal{C}(X)|\big) \leq H(P) + C$ for some constant C.

This distinction of weak universal codes if formal. General universalism is a strong property, but in many cases this might be irrelevant as the probability distribution may be much better known than merely monotonicity. In the case of FLAC's Golomb-Rice codes, weak universalism will suffice. In fact, Golomb-Rice codes are only weakly universal, but nevertheless practical for our use case.

In general, Huffman and arithmetic codes achieve better results than (weakly) universal codes whenever they can be applied [6]. Universal codes are therefore used only when this is not the case, for our purposes for instance, or when Huffman and arithmetic codes are inconvenient, as they require P to be specified to the decoder such that it can reconstruct $\mathcal{C}$. Transmitting these probabilities will in most scenarios lead to a sizeable overhead, from which universal codes do not suffer. Additionally, universal codes generally have greater computational efficiency than Huffman and arithmetic codes, especially when $\#\mathcal{A}$ gets progressively larger.

We conclude this section with a short statement about probability estimation. Most entropy codes rely on knowledge of the alphabet distribution P, which in most practical situations is not known beforehand. For this reason, encoders typically begin the construction of a suitable entropy code $\mathcal{C}$ by estimation of P. In such a scenario, we assume the availability of a sequence of random variables $X^N = (X_1, X_2, \cdots, X_N)$ which are *independent and identically distributed*, or IID, such that $X_i \sim P$ and

$$\mathbb{E}(X_k) = \mathbb{E}(X_1) \approx \frac{1}{N}\sum_{i=1}^{N} X_i \quad \text{and} \quad \mathrm{Var}(X_k) = \mathrm{Var}(X_1) \approx \frac{1}{N}\sum_{i=1}^{N} \big(X_i - \mathbb{E}(X_i)\big)^2.$$

From this parameters of a given a priori distribution can be estimated. Alternatively, P can be set to equal the found distribution in X^N exactly, as is done for some versions of Huffman and arithmetic coding.

3.2 An introduction to Golomb codes

As mentioned, FLAC uses *adaptive Golomb-Rice codes* to encode the residuals from the LPC step. They are a special case in the general family of Golomb codes, a class of symbol codes which are especially suited for exponentially or geometrically distributed symbols [15] [10]. We will discuss two examples of Golomb codes: the aforementioned Golomb-Rice codes and the exponential Golomb codes. The first will be described in particular detail, and the remainder of the chapter will concern it exclusively. To begin, we define the Golomb code as follows.

Definition 3.6 A *Golomb code* is an universal code characterized by an *index function f* which maps any x to

$$x \mapsto f(x) \coloneqq (i, r), \text{ with } \begin{cases} i \in \mathbb{N} \cup \{0\} & \text{the } \textit{set index}, \\ r \in \{1, 2, \ldots, R_i\} & \text{the } \textit{subindex}, \end{cases}$$

where i is written in *variable-length unary* and r in $\lceil \log R_i \rceil$ bit *fixed-length binary* (where log denotes the logarithm of base 2). In doing so, f defines the sets

$$S_i := \big\{x_r \mid f(x) = (i,r)\big\} = \{x_1, x_2, \ldots, x_{R_i}\}\,,$$
$$\text{with } R_i \text{ the largest value for } r \text{ given } i.$$

Note that in order to make this family of codes into prefix codes —from which unique decodability will follow— it is necessary to terminate the unary representation of i somehow. We will often write i as i consecutive `0`'s, so adding a single `1` in between i and r is sufficient.

The first and most important class of Golomb codes we will discuss are the Golomb-Rice codes, which are defined according to a parameter k.

Definition 3.7 *Golomb-Rice codes* with exponential parameter k are a form of Golomb codes where we define f_k for $x \in \mathbb{N} \cup \{0\}$ as

$$x \mapsto (i_x, r_x) = \Big(\lfloor x/2^k \rfloor, x \bmod 2^k\Big)\,, \text{ such that}$$
$$S_i = \Big\{x \in \mathbb{N} \cup \{0\} \mid n = i_x \cdot 2^k + r_x \text{ and } r_x < 2^k\Big\}\,.$$

Assuming the distribution of X is symmetric in 0 —which for the LPC residuals is reasonable as the audio nor the modelling step has preference for either negative or positive numbers— the best way to allow the usage of negative numbers as well is to simply add a single sign bit at the beginning of every encoded residual.

With this, we can explicitely define the Golomb-Rice codes.

Definition 3.8 We define the Golomb-Rice code $\mathcal{G}_k$ with parameter k as

$$\mathcal{G}_k\colon\ \mathbb{Z} \to \{0,1\}^+$$
$$\mathcal{G}_k(x) = \operatorname{sign}(x) i_x \mathtt{1} r_x, \quad \text{where } \big(i_x, r_x\big) = f_k\big(|x|\big).$$

Example 3.9 We provide some Rice-Golomb encoding examples for $k = 2$. Note that r_x is encoded in exactly $\lceil \log R_i \rceil = \log 2^k = k = 2$ bits, and that we add an extra `1` after i.

Number	**Sign**	(i_x, r_x)	**Binary**	**Full code**
$x = 0$	0	$(0,0)$	(,00)	0 100
$x = 1$	0	$(0,1)$	(,01)	0 101
$x = -2$	1	$(0,1)$	(,10)	1 110
$x = 4$	0	$(1,0)$	(0, 00)	0 0100
$x = -9$	1	$(2,1)$	(00, 01)	1 00101
$x = 15$	0	$(3,3)$	(000, 11)	0 000111

An advantageous property of Golomb-Rice codes is their efficient implementability in computer hardware and software. The computation of $i_x = \lfloor x/2^k \rfloor$ is simply the same as a *right-bitshift by k*, written as `|x| >> k`, while $r_x = x \bmod 2^k$ can be implemented as a *bitmask of the k least-significant bits*, written as `|x| &` $\underbrace{\mathtt{111\cdots1}}_{k \text{ bits long}}$.

The "adaptive" adjective of FLAC's method signifies that the parameter k is not fixed, but based on the distribution of the found residuals. Choosing the parameter optimally is not trivial, and therefore we will dedicate the next section to precisely this problem. First we will describe another popular Golomb code.

Definition 3.10 *Exponential Golomb codes* are a form of Golomb codes where we define f for $x \in \mathbb{N}$ as

$$x \mapsto (i, r) = \Big(\lfloor \log x \rfloor, x \bmod 2^{\lfloor \log x \rfloor}\Big), \text{ such that}$$
$$S_i = \Big\{x \in \mathbb{N} \mid 2^i \leq x < 2^{i+1}\Big\}.$$

Note that this method does not allow for `0` to be encoded, although this can be remedied easily by shifting every codeword up by one. Once again, we can add a sign bit to allow negative numbers.

When this method includes the additional `1`-bit between i and r, the encoding can be alternatively be described as:

1. Write in unary the amount of bits required to write x in binary, minus 1. Namely, write $\big(\lfloor \log x \rfloor + 1\big) - 1 = i$ bits.
2. Write x in binary of length $i + 1$.

This equivalence follows from the fact that $x \bmod x^{\lfloor \log x \rfloor}$ is nothing more than a *bitmask* over the first $\lfloor \log x \rfloor$ bits of x, where all higher order bits are discarded. In this case only the leading bit of x is removed, which will of course always be a `1`. This `1` is precisely the bit we have put between i and r.

Example 3.11 The exponential Golomb codes will look as follows. Note that indeed, the original binary representation of x is present in the encoding, and that the `0`-bits always indicate the binary length of x minus one.

Number	(i, r)	**Binary**	**Full code**
$x = 1$	$(0, 0)$	(,)	`1`
$x = 2$	$(1, 0)$	(,`0`)	`010`
$x = 3$	$(1, 1)$	(`0`, `1`)	`011`
$x = 4$	$(2, 0)$	(`0`, `00`)	`00100`
$x = 7$	$(2, 3)$	(`00`, `11`)	`00111`
$x = 8$	$(3, 0)$	(`000`, `000`)	`0001000`
$x = 14$	$(3, 6)$	(`000`, `111`)	`0001110`

The biggest difference with the linear Golomb-Rice codes is that r is not of fixed length, but will become larger as x becomes larger. In fact, we have that $\#S_{i+1} = 2 \cdot \#S_i$, and $S_0 = 1$. This idea can be expanded upon by letting each value of i correspond to a bit length of $i + k$ as apposed to just i, such that $\#S_0 = 2^k$ instead. In this way, larger numbers can be encoded more effectively with smaller values for i, however the effectiveness suffers for number closer to zero, since r will become relatively larger. Nonetheless, this gives rise to the so-called k-th order exponential Golomb codes, with which we conclude this section.

Definition 3.12 *k-th order exponential Golomb codes*, for $k \in \mathbb{N}$, are a form of Golomb codes where we define f_k for $x \in \mathbb{N}$ as

$$x \mapsto (i, r) = \Big(\big\lfloor \log\big(x + 2^k - 1\big)\big\rfloor - k, x \bmod 2^{\lfloor \log(x+2^k-1) \rfloor}\Big), \text{ such that}$$
$$S_i = \Big\{x \in \mathbb{N} \mid 2^{k+i} \leq x + 2^k - 1 < 2^{k+(i+1)}\Big\}.$$

Intuitively, adding $2^k - 1$ to x ensures the partitioning of $\mathbb{N}$ with $\{S_i\}$ is done as if every x is $2^k - 1$ higher, such that the partitioning —a result of the floor function— is essentially started at 2^k instead of at 1, with the first S_i of size 2^k as is the case for the normal exponential Golomb codes. This partitioning is shifted back to 1 in order to cover all of $\mathbb{N}$, such that indeed $\#S_0 = 2^k$.

Encoding can be done analogous to standard exponential Golomb encoding, which we can now alternatively describe as the 0-th order exponential Golomb code.

It should be noted that exponential Golomb codes are universal in additional to the weak universality that all described Golomb codes share, but we omit the proof here. Although exponential Golomb codes are generally not used in lossless audio compression formats, they are part of the hugely popular MPEG-4 Advanced Video Coding (AVC) standard for video compression, used by 91% of video industry developers as of September 2019 [26].

3.3 The Laplacian & Two-sided Geometric distributions

When we talk about optimality, it is always given the context of some distribution. A family of codes such as the Golomb-Rice codes $\{\mathcal{G}_i\}$ can be optimal for a class of distributions $\{P(p)\}$ if, given a particular distribution $P \in \{P(p)\}$, there exist a code $\mathcal{G}_k \in \{\mathcal{G}_i\}$ that is optimal. As we have found in the last section of chapter 2, the LPC residuals tend to behave conform the continuous Laplacian distribution. However in reality the residuals are discrete-valued, as is required for the Golomb-Rice codes and consequently our analysis of their optimality as well. The Laplacian's discrete counterpart is the *2-sided geometric distribution*, and this will be the symbol distribution we will consider for the remainder of this chapter. In this section specifically, we will derive and define this distribution, showing its close connection with the aforementioned Laplace distribution.

The Laplacian distribution is in turn derived from the exponential distribution, where the domain is extended to include all of $\mathbb{R}$, as apposed to only $\mathbb{R}_{\geq 0}$. It has two parameters: the *mean* μ around which the distribution is symmetric and the *scale* σ which influences the density around μ.

Definition 3.13 A random variable $Z \in \mathbb{R}$ has the *Laplacian distribution* if its probability density function is of the form

$$p_Z(x) = \frac{1}{2\sigma} e^{-\frac{1}{\sigma}|x-\mu|} = \begin{cases} \frac{1}{2} \cdot \frac{1}{\sigma} e^{-\frac{1}{\sigma}(x-\mu)} & \text{if } x \geq \mu \\ \frac{1}{2} \cdot \frac{1}{\sigma} e^{\frac{1}{\sigma}(x-\mu)} & \text{if } x < \mu, \end{cases}$$

for $\mu, \sigma \in \mathbb{R}$ and $\sigma > 0$. If this is the case, we write $Z \sim \text{Laplace}(\mu, \sigma)$.

The Laplacian distribution is exponential in both directions starting from μ, and can be constructed by taking an exponential distribution with parameter $1/\sigma$, mirroring it around the y-axis, adding another exponential distribution with parameter $1/\sigma$, and translating the whole ordeal to μ. Division by 2 brings the distribution to its completion, ensuring its integral over $\mathbb{R}$ is equal to 1. For our purposes, it is sufficient to limit ourselves to $\mu = 0$, although everything we discuss can be extended easily to general μ.

It is well known that the geometric distribution can be seen as the discrete counterpart to the exponential distribution, in the sense that the first can be derived from the second in a natural way. Namely, let $X \sim \text{Exp}(\lambda)$, and define $Y :=$

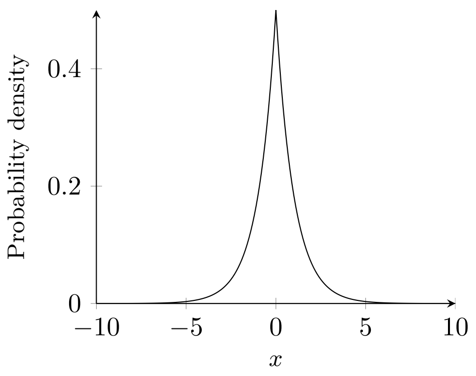


Figure 3.1: The Laplace probability density function for $\mu = 0$ and $\sigma = 1$.

$\lfloor X \rfloor$, such that $Y \in \{0, 1, 2, \ldots\}$. We write $p_X(x)$ for the probability density function of X, and find

$$\begin{aligned}\mathbb{P}(Y = k) = \mathbb{P}(k \leq X < k+1) &= \int_k^{k+1} p_X(x)\,\mathrm{d}x \\ &= \int_k^{k+1} \lambda e^{-\lambda x}\,\mathrm{d}x = \Big[-e^{-\lambda x}\Big]_k^{k+1} = \left(1 - e^{-\lambda}\right)\left(e^{-\lambda}\right)^k,\end{aligned}$$

which gives that $Y \sim \mathrm{Geom}\left(1 - e^{-\lambda}\right)$.

Just as the geometric distribution gives a discrete approximation of the exponential distribution, the *two-sided geometric distribution* gives a discrete approximation of the Laplace distribution. We provide a derivation. Let $Z \sim \mathrm{Laplace}(0, \sigma)$, then we define W as Z *rounded towards zero*, hence as

$$W := \begin{cases} \lfloor Z \rfloor & \text{if } Z \geq 0 \\ \lceil Z \rceil & \text{if } Z < 0 \end{cases}, \quad \text{which gives} \quad \mathbb{P}(W = k) = \begin{cases} \mathbb{P}\left(\lfloor Z \rfloor = k\right) & \text{if } k > 0 \\ \mathbb{P}\left(\lceil Z \rceil = k\right) & \text{if } k < 0 \\ \mathbb{P}\left(Z \in (-1, 1)\right) & \text{if } k = 0. \end{cases}$$

The symmetry of the Laplace distribution gives, if $k < 0$,

$$\mathbb{P}\left(\lceil Z \rceil = k\right) = \mathbb{P}\left(-\lfloor Z \rfloor = k\right) = \mathbb{P}\left(\lfloor Z \rfloor = -k\right) = \mathbb{P}\left(\lfloor Z \rfloor = |k|\right).$$

For positive k we can use that $2 \cdot Z$ is exponentially distributed on $[0, \infty)$. Note that $|k| = k$ if $k > 0$. It follows from the linearity of the integral and our previous derivation that for any $k \neq 0$,

$$\mathbb{P}\left(\lfloor Z \rfloor = |k|\right) = \tfrac{1}{2}\left(1 - e^{-\frac{1}{\sigma}}\right)\left(e^{-\frac{1}{\sigma}}\right)^{|k|}.$$

Now only $W = k = 0$ remains, which is somewhat of a special case. Both the left and right side of the distribution add to its probability, so we find

$$\begin{aligned}\mathbb{P}\left(Z \in (-1, 1)\right) &= \mathbb{P}\left(\lceil Z \rceil = 0 \text{ or } \lfloor Z \rfloor = 0\right) = \mathbb{P}\left(\lceil Z \rceil = 0\right) + \mathbb{P}\left(\lfloor Z \rfloor = 0\right) \\ &= \left(1 - e^{-\frac{1}{\sigma}}\right)\left(e^{-\frac{1}{\sigma}}\right)^0 = 1 - e^{-\frac{1}{\sigma}}.\end{aligned}$$

Putting everything together, we arrive at

$$\mathbb{P}(W = k) = \begin{cases} \frac{1}{2}\left(1 - e^{-\frac{1}{\sigma}}\right)\left(e^{-\frac{1}{\sigma}}\right)^{|k|} & \text{if } k \neq 0 \\ 1 - e^{-\frac{1}{\sigma}} & \text{if } k = 0. \end{cases}$$

This gives us a rigorous basis for defining the distribution.

Definition 3.14 A random variable $W \in \mathbb{Z}$ has the *2-sided geometric distribution* if its probability density function is of the form

$$\mathbb{P}(W = k) = \begin{cases} \frac{1}{2}p(1-p)^{|k|} & \text{if } k \neq 0 \\ p & \text{if } k = 0 \end{cases}$$

for $p \in (0, 1]$. If this is the case, we write $W \sim 2\mathrm{Geom}(p)$.

Note that our derivation resulted in $p = 1 - e^{-\frac{1}{\sigma}}$, with σ the parameter for the original Laplacian distribution. Furthermore, one can quickly see that

$$\mathbb{P}\left(|W| = k\right) = p(1-p)^k \quad \text{if } k \geq 0,$$

such that $|W| \sim \mathrm{Geom}(p)$. This is advantageous computationally speaking: for $W_1, W_2, \dots, W_N \overset{\text{IID}}{\sim} 2\mathrm{Geom}(p)$, we can easily approximate p with

$$\frac{1-p}{p} = \mathbb{E}\left(|W_1|\right) \approx \frac{1}{N}\sum_{n=1}^{N}|W_n|.$$

3.4 Optimality of Golomb-Rice codes

Theorem 3.4 tells us that any encoding scheme is optimal if the average codeword length approaches the Shannon entropy of the symbol distribution. As mentioned, in practise this often comes down to "really close" to the entropy, which will be the case here as well. For $W \sim 2\mathrm{Geom}(p)$ we will construct the optimal Golomb-Rice code $\mathcal{G}_k$, and subsequently demonstrate an upper bound $H(W) + 0.6$ computationally, followed by a rigorous proof of a more crude bound.

As mentioned, the choice for $W \sim 2\mathrm{Geom}(p)$ is motivated by section 2.5, where the distribution describes the residuals of the LPC step. Subsequently, our result will require two main ingredients.

1. The entropy of a 2-sided geometric distribution. (Lemma 3.15)
2. The average codeword length of Golomb-Rice codes of $W \sim 2\mathrm{Geom}(p)$ given a Rice parameter k. (Lemma 3.16)

Lemma 3.15 Let $W \sim 2\mathrm{Geom}(p)$ with $p \in (0,1)$. Then the entropy of W is equal to

$$H(W) = 1 + \log\left(\frac{1-p}{p}\right) - p - \tfrac{1}{p}\log(1-p).$$

Proof By the symmetry of the distribution we have

$$\begin{aligned} H(W) &= \sum_{k=-\infty}^{\infty} \mathbb{P}(W=k)\log\frac{1}{\mathbb{P}(W=k)} \\ &= \mathbb{P}(W=0)\log\frac{1}{\mathbb{P}(W=0)} + 2\sum_{k=1}^{\infty}\mathbb{P}(W=k)\log\frac{1}{\mathbb{P}(W=k)}, \end{aligned}$$

in which we can substitute the distribution's definition to find

$$\begin{aligned} H(W) &= p\log\frac{1}{p} + 2\sum_{k=1}^{\infty}\tfrac{1}{2}p(1-p)^{|k|}\log\frac{1}{\tfrac{1}{2}p(1-p)^{|k|}} \\ &= -p\log p + p\sum_{k=1}^{\infty}(1-p)^k\big(\log 2 - \log p - k\cdot\log(1-p)\big) \\ &= -p\log p + \Big(p(1-\log p)\sum_{k=1}^{\infty}(1-p)^k\Big) - \Big(p\log(1-p)\sum_{k=1}^{\infty}(1-p)^k k\Big). \end{aligned}$$

We make a quick excursion to geometric series. As $1-p<1$, we know that

$$\sum_{k=0}^{\infty}(1-p)^k = \frac{1}{1-(1-p)} = \frac{1}{p}.$$

We can take out the constant term to find the first series, and differentiating with respect to p on both sides to find the other. It follows that

$$\begin{aligned}
H(W) &= -p\log p + \Big(p(1-\log p)\big(\tfrac{1}{p}-1\big)\Big) - \Big(p\log(1-p)\big(\tfrac{1-p}{p^2}\big)\Big)\\
&= -p\log p + \big(1-\log p - p + p\log p\big) - \tfrac{1-p}{p}\log(1-p)\\
&= 1 - p - \log p - \tfrac{1}{p}\log(1-p) + \log(1-p)\\
&= 1 + \log\Big(\frac{1-p}{p}\Big) - p - \tfrac{1}{p}\log(1-p). \qquad \square
\end{aligned}$$

Before we continue with lemma 3.16, recall that we write $\big|\mathcal{G}_k(x)\big| \in \mathbb{N}$ as the length of the Golomb-Rice codeword for x. This bitstring is the concatenation of the unary i_x, binary r_x for $|x|$ and the sign bit, as well as the bit to terminate i_x.

Lemma 3.16 With exponential variable k, the average encoding length of a random variable $W \sim 2\mathrm{Geom}(p)$ using Golomb-Rice encoding satisfies

$$\mathbb{E}\Big(\big|\mathcal{G}_k(W)\big|\Big) = 1 + k + \frac{1}{1-(1-p)^{2^k}}.$$

Proof Starting with breaking up $\big|\mathcal{G}_k(\cdot)\big|$ into its components and substituting their definitions as found in 3.7 we get

$$\begin{aligned}
\mathbb{E}\Big(\big|\mathcal{G}_k(W)\big|\Big) &= \sum_{x=-\infty}^{\infty} \big|\mathcal{G}_k(x)\big|\,\mathbb{P}(W=x) = \sum_{x=-\infty}^{\infty} \big(1 + i_x + 1 + \lceil\log(R_{i_x})\rceil\,\big)\,\mathbb{P}(W=x)\\
&= \sum_{x=-\infty}^{\infty} \big(1 + \big\lfloor |x|/2^k\big\rfloor + 1 + k\big)\,\mathbb{P}(W=x),
\end{aligned}$$

which we can rewrite to

$$\begin{aligned}
\mathbb{E}\Big(\big|\mathcal{G}_k(W)\big|\Big) &= (2+k)p + \sum_{x=1}^{\infty}\big(2 + \lfloor x/2^k\rfloor + k\big)p(1-p)^x\\
&= (2+k)p + p(2+k)\sum_{x=1}^{\infty}(1-p)^x + p\sum_{x=1}^{\infty}\lfloor x/2^k\rfloor(1-p)^x\\
&= (2+k)p + p(2+k)\big(\tfrac{1}{p}-1\big) + p\sum_{x=1}^{\infty}\lfloor x/2^k\rfloor(1-p)^x\\
&= 2 + k + p\sum_{x=1}^{\infty}\lfloor x/2^k\rfloor(1-p)^x.
\end{aligned}$$

Dealing with the floor with in the geometric series requires some extra work. We find

$$\begin{aligned}
\sum_{x=1}^{\infty}\lfloor x/2^k\rfloor(1-p)^x &= \sum_{y=1}^{\infty}\sum_{x=0}^{2^k-1} y(1-p)^{2^k y + x} = \sum_{y=1}^{\infty} y(1-p)^{2^k y}\sum_{x=0}^{2^k-1}(1-p)^x\\
&= \sum_{y=1}^{\infty} y(1-p)^{2^k y}\cdot\frac{1-(1-p)^{2^k}}{1-(1-p)} = \frac{1-(1-p)^{2^k}}{p}\sum_{y=1}^{\infty} y\Big((1-p)^{2^k}\Big)^y\\
&= \frac{1-(1-p)^{2^k}}{p}\cdot\frac{1}{\big(1-(1-p)^{2^k}\big)^2}(1-p)^{2^k} = \frac{1}{p}\cdot\frac{(1-p)^{2^k}}{1-(1-p)^{2^k}},
\end{aligned}$$

with which we arrive at our final result. $\square$

Our goal is now to, given p, *minimise* $\mathbb{E}\big(|\mathcal{G}_k(W)|\big)$ over k. For this, we will choose k by way of an *educated guess* based on p, which we will find to in fact be optimal. We provide three notable such guesses, and our upcoming demonstration of optimality is based on a particular one. In the case of LPC these guesses are used to encode the residual sets, and p can be estimated based on this set.

All guesses are based on the idea that we want to choose k such that with probability $1/2$ any sample $|W|$ lies within the range $S_0 = \{0, 1, \dots, 2^k - 1\}$, or in other words, such that $\mathbb{P}\big(|W| \in S_0\big) = 1/2$. Too low values of k will lead to the increased use of the expensive unary bits i_x, and conversely too high values of k will result in a lot of "wasted" high-order bits in the binary r_x. Since both increase proportional to the logarithm of $|W|$ and $|W|$ is geometrically distributed, these wasted and unary bits carry, intuitively speaking, equal weight. As a result, neutralizing the probability around $1/2$ should balance them optimaly.

We now begin with the three guesses.

The Laplace Guess

The *Laplace guess* is based the above principle, using an input variable satisfying the Laplacian distribution (definition 3.13). It is included here as the LPC residuals are traditionally described as being Laplacian distributed, instead of the more appropriate 2-sided geometric distribution. It is given by

$$\hat{k}_{\text{laplace}} = \log\Big(-\ln(2)\frac{1}{\ln(1-p)}\Big), \tag{3}$$

and rounding this number to the final guess k_{laplace}.

The derivation is as follows. Take $Z \sim \text{Laplace}(0, \sigma)$, then we find

$$1/2 = \mathbb{P}\big(|Z| \in S_0\big) = 2\int_0^{2^k} \frac{1}{2\sigma} e^{-\frac{1}{\sigma}x}\,\mathrm{d}x = \Big[-e^{-\frac{1}{\sigma}x}\Big]_0^{2^k} = 1 - e^{-\frac{1}{\sigma}2^k}$$

Which implies that $1/2 = e^{-\frac{1}{\sigma}2^k}$ such that $\ln(2) = \frac{1}{\sigma}2^k$, and finally $k = \log(\ln(2)\sigma)$.

Substituting $p = 1 - e^{-\frac{1}{\sigma}} \implies \sigma = -\frac{1}{\ln(1-p)}$ as we found in our derivation in the previous section, we arrive at (1).

The 2-sided geometric guess

This guess is similar to the previous, but uses $W \sim 2\text{Geom}(p)$ instead. It is given by

$$\hat{k}_{\text{2geom}} = -\log\log\Big(\frac{1}{1-p}\Big), \quad \text{rounded to } k_{\text{2Geom}}. \tag{4}$$

This can be easily derived as follows.

$$\begin{aligned} 1/2 = \mathbb{P}\big(|W| \in S_0\big) = \mathbb{P}\big(|W| < 2^k\big) &= \sum_{x=0}^{2^k-1} \mathbb{P}\big(|W| = x\big) = \sum_{x=0}^{2^k-1} p(1-p)^x \\ &= p\frac{(1-p)^{2^k}}{1-(1-p)} = (1-p)^{2^k} \\ \implies \quad 2^k \log(1-p) = \log(1/2) = -1 \quad &\implies \quad k = -\log\log\big(\tfrac{1}{1-p}\big). \end{aligned}$$

The absolute average guess

This guess is somewhat different from the previous two. It is motivated by the fact that, for $W \sim 2\text{Geom}(p)$,

$$\mathbb{E}\left(|W|\right) = \frac{1-p}{p}, \quad \text{since } |W| \sim \text{Geom}(p).$$

The bits required to write this number in binary minus one will be our guess for k_{average}, such that once again about half of the random samples will fit in S_0, although this time with greater approximation. The guess then becomes

$$k_{\text{average}} = \left\lfloor \log\left(\frac{1-p}{p}\right) \right\rfloor. \tag{5}$$

This is the guess that FLAC uses to determine its Rice parameter. And due to its simplicity, it will also be the guess for which we will prove optimality.

Note that for $p > 1/2$ the expected value of $W \sim 2\text{Geom}(p)$ is close to 0 to such a degree that taking $k = 0$ will result in the best possible Golomb-Rice code. Therefore in addition to the guesses provided above, every guess takes $k = 0$ when $p > 1/2$, which has the additional advantage of eliminating any negative guesses which result from the given specifications.

We will use lemma 3.16 to analyse the performance of these guesses, while comparing against the entropy as given in lemma 3.15. First, we provide a computational comparison as found in the figure below. We find, although the absolute average guess performs worse, all guesses are within a range of approximately 0.6 of the $H(W)$ for $p < 0.5$. For $p \geq 0.5$ the codes become significantly worse —a result of the codeword length having a minimal size of 2 bits— although in practise values of p are encountered only very rarely.

Of course, a bound of 0.6 may not be good in all circumstances, however an important aspect is that the optimality increases as p approaches 0. Additionally, when codeword length become increasingly long, this bound becomes relatively better.

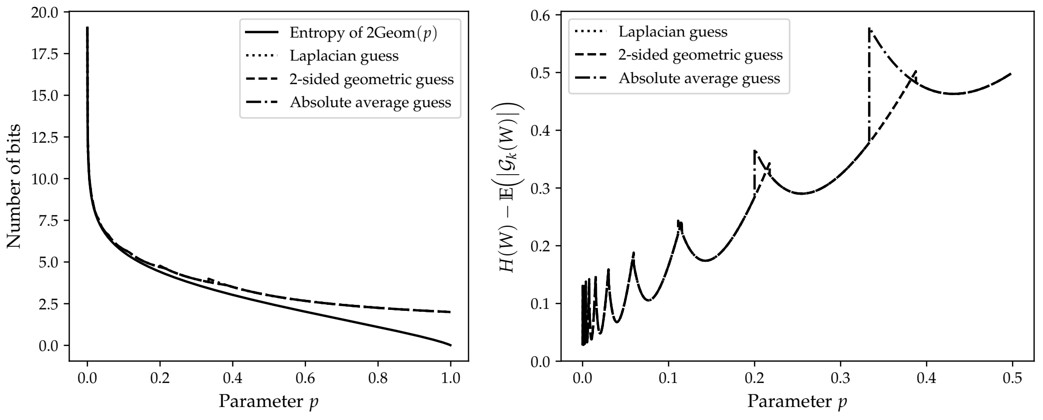


Figure 3.2: Left: The expected codeword lengths for the three guesses, together with the entropy of the 2-sided geometric distribution. Right: The distance from the entropy of the expected codeword lengths, zoomed in to $p \leq 1/2$. The Laplacian and 2-sided geometric guesses are equal, and perform better than the absolute average guess.

To finish this chapter, we prove the following crude bound for the k_{average} specifically.

Theorem 3.17 (Optimality of Golomb-Rice codes) For $W \sim 2\text{Geom}(p)$ and $k_{\text{average}} = \left\lfloor \log\left(\frac{1-p}{p}\right) \right\rfloor$ if $p \leq 1/2$ and $k_{\text{average}} = 0$ otherwise, the average encoding length of the Golomb-Rice code $\mathcal{G}_{k_{\text{average}}}$ satisfies

$$\mathbb{E}\big(\mathcal{G}_{k_{\text{average}}}(W)\big) \leq H(W) + \frac{1}{1-\frac{1}{\sqrt{2}}} - \left(\frac{3}{4} + \frac{13}{16\log(2)} - \frac{3\log(2)}{4}\right) \approx H(W) + 2.012.$$

Proof As mentioned, for this proof we will consider the absolute average guess as found in expression (3). To start, we note that for $p > 1/2$ we have $-\log(1-p) > 1$, hence

$$\mathbb{E}\big(\mathcal{G}_0(W)\big) \overset{\text{lemma 3.16}}{=} 1 + \tfrac{1}{p} = 2 + \left(\tfrac{1}{p} - 1\right) < 2 - \left(\tfrac{1}{p} - 1\right)\log(1-p).$$

So by lemma 3.15, $\mathbb{E}\big(\mathcal{G}_0(W)\big) < 1 - p + \log\frac{1}{p} + 2 - \left(\frac{1}{p} - 1\right)\log(1-p) = H(W) + 2$.

The proof for $p \leq 1/2$ consists of two parts. First off all, we have that

$$\mathbb{E}\big(\mathcal{G}_{k_{\text{average}}}(W)\big) \leq 1 + \log\left(\tfrac{1-p}{p}\right) + \frac{1}{1-(1-p)^{\frac{1-p}{2p}}},$$

which follows from the substitution of (3) in the expression of lemma 3.16, together with the fact that

$$1 + \left\lfloor \log\left(\tfrac{1-p}{p}\right)\right\rfloor + \frac{1}{1-(1-p)^{2^{\left\lfloor \log\left(\frac{1-p}{p}\right)\right\rfloor}}} \leq 1 + \log\left(\tfrac{1-p}{p}\right) + \frac{1}{1-(1-p)^{2^{\log\left(\frac{1-p}{p}\right)-1}}}.$$

It must be noted that this bound loses a little less than 1 in its precision as opposed to the true bound, although for simplicity we won't refine it any further here.

The second part of the proof will be the estimation of part of this bound —note that $1 + \log\left(\frac{1-p}{p}\right)$ is already a component of $H(W)$—, namely we will show

$$\frac{1}{1-(1-p)^{\frac{1-p}{2p}}} \leq M - p - \tfrac{1}{p}\log(1-p), \qquad \text{for some } M \in \mathbb{R}\,.$$

Starting with the right hand side, we find by the Taylor expansion of $-\log(1-p)$ that

$$-p - \tfrac{1}{p}\log(1-p) = -p + \frac{1}{\ln(2)}\sum_{n=0}\frac{p^n}{(n+1)} \geq -p + \frac{1}{\ln(2)}\Big(1 + \frac{p}{2} + \frac{p^2}{3}\Big),$$

which attains a minimum at $p = \ln(2)\frac{3}{2} - \frac{3}{4}$ and consequently demonstrates a lower bound $M_1 \leq -p - \frac{1}{p}\log(1-p)$. Conversely, the left hand side is monotonically increasing, such that

$$\frac{1}{1-(1-p)^{\frac{1-p}{2p}}} \leq \frac{1}{1-\left(1-\frac{1}{2}\right)^{\frac{1-1/2}{2/2}}} = \frac{1}{1-\frac{1}{\sqrt{2}}} =: M_2.$$

To show that the left hand side is indeed monotonous, we note that

$$\frac{1}{1-(1-p)^{\frac{1-p}{2p}}} = \frac{1}{1-e^{\ln(1-p)\frac{1-p}{2p}}}.$$

By the monotonicity of $\frac{1}{1-e^x}$ it remains to show that $\ln(1-p)\frac{1-p}{2p}$ is monotone, which is clear since we can bound its derivative as

$$-\frac{1-p}{(1-p)2p} - \ln(1-p)\frac{1}{2p^2} = \frac{-p-\ln(1-p)}{2p^2} \geq \frac{-p+1}{2p^2} > 0.$$

Putting everything together, we have

$$\begin{aligned}
\mathbb{E}\big(\mathcal{G}_{k_{\text{average}}}(W)\big) &\leq 1 + \log\left(\tfrac{1-p}{p}\right) + \frac{1}{1-(1-p)^{\frac{1-p}{2p}}} \\
&\leq 1 + \log\left(\tfrac{1-p}{p}\right) + M_2 \\
&\leq 1 + \log\left(\tfrac{1-p}{p}\right) + M_2 - M_1 - p - \frac{1}{p}\log(1-p) \\
&= H(W) + M_2 - M_1.
\end{aligned}$$

Where $M_2 - M_1 = \frac{1}{1-\frac{1}{\sqrt{2}}} - \left(\frac{3}{4} + \frac{13}{16\log(2)} - \frac{3\log(2)}{4}\right) \approx 2.012.$ □

4 Integrating into the FLAC audio compression format

This chapter will come in two parts. First, we give a technical description of the FLAC format and provide details on the integration of the four algorithms for computation of the LPC parameters discussed in chapter 2 into the codec. Secondly, we provide an analysis and comparison of the compression performance, consisting of the encoding speed and compression ratio, of these algorithms. Since all algorithms integrate into the FLAC format specification, decoding speeds will be equal and its analysis is therefore excluded.

We implement only a subset of the full FLAC feature set, however the further possible enhancements are secondary and apply to each algorithms equally well. Moreover, their exclusion does not hinder a fair comparison between the linear prediction algorithms. One should be aware that for this reason, the FLAC reference encoder will be superior to the presented algorithms. This holds true for computational factors as well, as the our comparison use algorithms with virtually no optimisation. Since this applies for all algorithms we will compare in this chapter, the provided results concerning compression time are worthwhile but should be taken with a grain of salt.

For the exact and complete specifications we refer to the official format specification [4] [1], which should be consulted for the exact bytecode layout and formatting, as well as for details about metadata inclusion. For information about the FLAC reference encoder and its source code, see [7]. Our implementation can be found over at `https://github.com/amsqi/adaptive-flac`.

4.1 FLAC format overview

As mentioned, FLAC follows the general encoding scheme presented in the introduction, consisting of the blocking step, the interchannel decorrelation, the linear predictive coding and finally the —in this case Golomb-Rice— entropy coder. FLAC defines the distinction between *blocks*, which partition the original audio signal, and *frames*, which partition the encoded file, each frame representing exactly one block. Both blocks and frames are further subdivided in subblocks and subframes respectively, each subblock and subframe pertaining to exactly one audio channel. Channels are encoded independently, however they may not represent the original channels exactly after the interchannel decorrelation step. Blocksize is independent from channel count, so as an example for stereo audio —which has 2 channels— blocks of length 4096 would contains 8192 samples audio data.

Every FLAC file begins with a header, consisting of one or more metadata blocks which can of instance contain information about the file author, the cuesheet or cover art, although we will omit most details here. One metadata block is mandatory, the `STREAM_INFO` block which contains general information about the upcoming encoded audio data, such as the number of channels and the bitdepth of the encoded audio file. At a high level, an FLAC file will have the following layout.

Content	**Description**
`"fLaC"`	Marks the start of the FLAC file. Letters are encoded in ASCII.
`STREAM_INFO`	The mandatory metadata block with general file properties.
`METADATA_BLOCK*`	Zero or more metadata blocks.
`FRAME+`	One or more encoded frames.

Table 4.1: High-level FLAC data stream specification

The `STREAM_INFO` block encodes the minimum and maximum blocksize and frame size used, the sample rate, the amount of channels, the bits per sample (the sample bitdepth) of the original audio signal, the total length of the audio signal in samples and a MD5 signature of the entire stream for error detection. Omitting the metadata header, the block is structured as follows.

Content	Bits	Description
`MIN_BLOCKSIZE`	16	Minimum blocksize used in samples, must be 16 or higher.
`MAX_BLOCKSIZE`	16	Maximum blocksize used in samples.
`MIN_FRAMESIZE`	24	Minimum frame size used in samples, 0 implies unknown.
`MAX_FRAMESIZE`	24	Maximum frame size used in samples, 0 implies unknown.
`SAMPLE_RATE`	20	Sample rate in Hertz.
`CHANNEL_COUNT`	3	Number of channels, minus 1.
`BITDEPTH`	5	Bits used per sample in the original audio data, minus 1.
`LENGTH`	36	Total length of audio data in samples.
`MD5_SIGNATURE`	128	MD5 signature of the unencoded audio data.

Table 4.2: The `STREAM_INFO` metadata block

As is implied by this specification, the FLAC format allows encoding of variable blocksizes, even though almost all encoders do not utilize it. This creates the possibility for the adaptive linear prediction algorithms to be used in a FLAC encoder, with standard FLAC decoders being able to decode the encoded data. By the 16 bit limitation, we note that FLAC supports blocksizes between 16 and $2^{16} - 1 = 65535$ samples long.

One important consideration FLAC makes is *streaming compliance*, which means decoding should be possible starting at any frame of the encoded audio such that the FLAC data can be send of a network connection without desynchronisation becoming a large issues, given general information provided in the `STREAMINFO` block from the header is known. As a result, some general information about the audio stream is included in every frame, in addition to the size of the encoded block. Similar to the complete audio stream, a frame consists of the following data.

Content	Description
`FRAME_HEADER`	General info about the frame.
`SUBFRAME+`	A subframe for every channel.
`0*`	0-padding to ensure byte alignment.
`FRAME_FOOTER`	Contains a CRC-16 of entire encoded frame.

Table 4.3: High-level frame specification

Cyclic redundancy checks (CRC's) are used to guarantee no errors have occurred during transmission of the encoded data. They are computed with polynomial division over the entire encoded frame —in this case with the polynomial $x^{16} + x^{15} + x^2 + x^0$— such that after the decoder has done the same, it can compare the two values and verify equality. We will not provide details on how exactly they function here, although since many variations exists pseudo-code for their computation is included in the appendix.

The frame header is structured similarly to the `STREAM_INFO` block. Since bits are expensive, the information is given as an index in a list of common values, with the option to specify any value at the end of the header if necessary. We only provide a quick overview here, for the full description, and table definitions, one should consult the format specification [4] [1].

Content	Bits	Description
`11111111111110`	14	Sync code.
`0`	1	Mandatory bit.
`BLOCK_STRATEGY`	1	`1` means variable-size blocks, `0` means fixed-size.
`BLOCKSIZE`	4	Table index for the encoded blocksize.
`SAMPLE_RATE`	4	Table index for the sample rate.

`CHANNEL_ASSIGNMENT`	4	Table index for the channel assignment.
`BITDEPTH`	3	Table index for the bitdepth.
`0`	1	Mandatory bit.
Frame position	8-56	Sample number or frame number, UTF-8 encoded.
Blocksize	0/8/16	Optional specification of the blocksize (minus 1).
Sample rate	0/8/16	Optional specification of the sample rate.
`CRC-8`	8	CRC-8 of the frame header, with polynomial $x^8 + x^2 + x^1 + x^0$

Table 4.4: The frame header

Wether the sample number of frame number is encoded depends on the blocking strategy, where for variable-size blocks the sample number is recorded, and otherwise the frame number. UTF-8 encoded means the number is encoded as UTF-8 encodes characters. Once again, we provide the details and pseudo-code for this method in the appendix.

4.1.1 Interchannel decorrelation

Before we specify the subframe layout, we shortly discuss the `CHANNEL_ASSIGNMENT` from the frame header specification, which has everything to do with the channel decorrelation step. For a stereo audio stream, the first thing an encoder does is the decorrelation of the two audio channels by encoding their common characteristics in one channel, and the different in the other, such that the correlation inherent between the channels is removed and the magnitude of the second channel is reduced. There are multiple approaches, such as the mid-side channel assignment we discussed in the introduction. Taking the stereo channel $X_n^{(\text{left})}$ and $X_n^{(\text{right})}$, we define

$$X_n^{(\texttt{mid})} := \frac{X_n^{(\texttt{left})} + X_n^{(\texttt{right})}}{2} \quad \text{and} \quad X_n^{(\texttt{side})} := X_n^{(\texttt{left})} - X_n^{(\texttt{right})}.$$

It is also possible to leave the left or right channel intact, and define the side channel as respectively

$$X_n^{(\texttt{side})} := X_n^{(\texttt{left})} - X_n^{(\texttt{right})} \quad \text{or} \quad X_n^{(\texttt{side})} := X_n^{(\texttt{left})} - X_n^{(\texttt{right})}.$$

The `CHANNEL_ASSIGNMENT` table then looks as follows.

Value	Description
`0000-0111`	Left-right assignment. Index is #channels minus one.
`1000`	Left-side channel assignment.
`1001`	Right-side channel assignment.
`1010`	Mid-side channel assignment.
`1011-1111`	Reserved.

Table 4.5: The `CHANNEL_ASSIGNMENT` value table

All channels from the channel assignment are subsequently encoded independently, each in their own subframe and in the same order as in the description.

4.1.2 Linear prediction and more

With all block properties specified in the frame header, the subframe only needs to specify the encoding method and provide the associated encoded data. Besides linear predictive coding, FLAC allows for three other types of subframes: a constant value, a fixed polynomial predictor and a verbatim block. Every subframe is made up of a header and subframe body, of which there is one for all four subframe types.

Content	Description
SUBFRAME_HEADER	Specifies the model type and order.
One of the following:	
SUBFRAME_CONSTANT	Encodes block with a single constant value (often 0).
SUBFRAME_FIXED	Encodes block with a fixed polynomial predictor.
SUBFRAME_LPC	Encodes block with LPC.
SUBFRAME_VERBATIM	Encodes block uncompressed.

Table 4.6: High-level subframe specification

Since we will be analysing the LPC encoding exclusively, we will only introduce its specification. For an encoded order p, we need to store

- The LPC coefficient set $\{a_j\}$ of size p in transversal form. There are stored using *vector quantization*, which encodes the parameters as binary integer values $\{\hat{a}_j\}$ in *precision* amount of bits together with a *shift value* s, such that $a_j \approx \hat{a}_j$ >> s, where >> denotes the binary shift operation.
- The p *warm-up* values. The first p values of the block uncompressed, which are required before being able to compute the first linear prediction.
- The residuals set, which we will specify in the next section.

The subframe then looks as follows.

Content	Bits	Description
WARMUP	BITDEPTH $\cdot\, p$	The warm-up samples, uncompressed
PRECISION	4	Vector quantization precision minus 1.
SHIFT	5	Vector quantization shift value (signed).
COEFFICIENTS	PRECISION $\cdot\, p$	The vector quantized coefficient set.
RESIDUAL	?	The encoded residuals.

Table 4.7: The LPC subframe

The FLAC reference encoder uses the modified Levinson-Durbin algorithm as presented in algorithm 2.18 for calculation of the LPC coefficients. Since the decoder needs to be able to reproduce the coefficient set exactly, for residual computation the vector quantized coefficients $\hat{a}_j$ >> s are used. Pseudo-code for computation of the precision and shift values is included in the appendix.

4.1.3 Residual encoding with Golomb-Rice codes

For the residual encoding using the Golomb-Rice codes it is necessary to provide the Rice parameter along with the encoded residuals. For FLAC, two things are notable, firstly the Rice parameter is to be encoded with either 4 bits or 5 bits, and secondly that further (equidistant) partitioning of the block is supported, where each section is encoded with its own Rice parameter. Like this, the Rice-parameter can be more accurate to the residuals of the given section, which improves the compression in many situations.

The residual header is simple. It contains a 2 bit long value to indicate whether the Rice parameter is encoding with 4 or 5 bits, followed by a 4 bit long *partition order* m. Then, 2^m so-called RICE_PARTITIONs follow, which look like this.

Content	Bits	Description
PARAMETER	4 or 5	The Rice parameter k.
Residuals	?	BLOCK_SIZE$/2^m$ residuals encoded with k. (BLOCK_SIZE$/2^m - p$ if this is the first section)

Table 4.8: The RESIDUAL specification

FLAC stores the encoded residuals exactly as discussed in chapter 3, apart from the fact that the sign bit is stored at the *end* of each encoded residuals instead of the front.

4.2 Experiments and comparisons

Once acquainted with the format specification, the integration of the four linear prediction algorithms should be straightforward. The algorithms in question are the modified Levinson-Durbin algorithm (algorithm 2.18), the covariance Burg algorithm (algorithm 2.29), as well as the two new adaptive algorithms, the adaptive Burg algorithm (algorithm 2.30) and the autocorrelation adaptive Burg algorithm (subsection 2.4.2). In this section we will investigate the performance of these approaches within the FLAC format.

We will regard *compression ratio* and *execution time* as the main points of concern in the upcoming experiments. Since decoding is fixed by the FLAC format specification, this aspect —which we did mention in the introduction as central for lossless compression— is of no importance to us in this comparison and has been withdrawn from the evaluation.

Once again, we want to highlight that the full FLAC specification is not employed, namely that we will not make use of optimal order detection, fixed predictors, constant subframe blocks and residual partitioning. The comparison will be between the algorithms by themselves, so there is no particular need for the inclusion of these additional, LPC-independent, improvements. Nevertheless, one should realize that for this reason the shown results are no match for the FLAC reference encoder, both in terms of compression ratio and compression time.

Lastly, Burg algorithm and its two adaptive derivatives have further computational improvements which have not been implemented, as given in [19], which one should keep in mind during the upcoming findings on execution time.

Now to begin with the experiments, for which we have made the following selection of music, chosen to vary in musical contents and instrumentation. We will study the six genres one-by-one, and highlight the differences. For better comparisons across and within genres, each track is represented by a fragment exactly 30 seconds long.

Genre	**Album**	**Artists**	**Tracks**
Piano	Ghibli piano collection	Joe Hisaishi, cover by Carolyn	17
Rock	Your Name OST (selection)	RADWIMPS	5
Electronic jazz	Black Box	aivi & surasshu	11
Electronic	Crypt of the Necrodancer remixed	Jake Kaufman	20
Orchestral	Star Wars: The Force Awakens	John Williams	23
Cuban	Lost and found	Buena Vista Social Club	12

Table 4.9: The audio data used for the comparison. For all analyses, 30 second excerpt of all songs were used.

For each genre, the compression ratio —the compressed file size divided by the original file size— and execution time of the encoding was averaged of all tracks, repeated for LPC orders 4, 6, 8, 10, 12 and 14. The two fixed-blocksize algorithms use a blocksize of 4096 samples, and for the adaptive algorithms we set the detection parameters as $N_{\text{range}} = 2048$ and $r = 1.01$. Both use a smaller fixed order of 4 for detection as illustrated in subsection 2.4.2.

We begin with the "Piano" genre in figure 4.1, and immediately notice that, as predicted, the two adaptive algorithms suffer from worse computation times compared to the standard algorithms, which will be a reoccurring result. The autocorrelation method is significantly faster however, and surprisingly achieves better compressions ratios as well. The covariance

Compression ratio and execution time for genre “Piano”

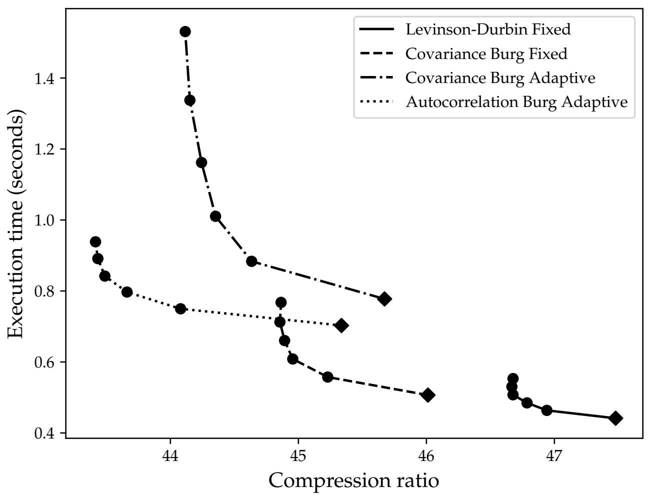


Figure 4.1: The compression ratio (compressed file size divided by original file size) and computation time of the four algorithm, for LPC orders 4, 6, 8, 10, 12 and 14, starting at the diamond and continuing along sequentially along the line. Uses the “Piano” dataset from table 4.9.

Burg algorithm performs better than the modified Levinson-Durbin algorithm, and in this case specifically, the adaptive algorithms do even better, although we emphasize that the difference is only a few percentages.

Additionally, the Levinson-Durbin, covariance Burg and autocorrelation adaptive Burg methods scale considerably less when the order p increases compared to the covariance adaptive Burg algorithm. This is in accordance with our theoretical results on the complexity of these algorithms, where the latter was the only algorithm with complexity $\mathcal{O}(p^2N)$, as opposed to a complexity of $\mathcal{O}(pN)$.

Compression ratio and execution time for genre “Rock”

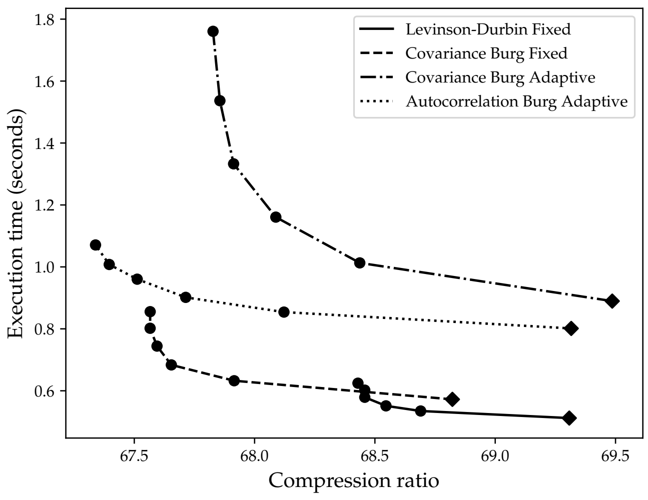


Figure 4.2: The compression ratio (compressed file size divided by original file size) and computation time of the four algorithm, for LPC orders 4, 6, 8, 10, 12 and 14, starting at the diamond and continuing along sequentially along the line. Uses the “Rock” dataset from table 4.9.

For “Rock” the compression ratio is poor across all algorithms, however the same patterns as discussed previously appear. The adaptive algorithms are worse for the lowest orders, although they improve considerably and the autocorrelation even overtakes the two fixed-blocksize algorithms in terms of compression ratio.

The following two genres show unfavorable results for the adaptive algorithms.

Compression ratio and execution time for genre “Electronic Jazz”

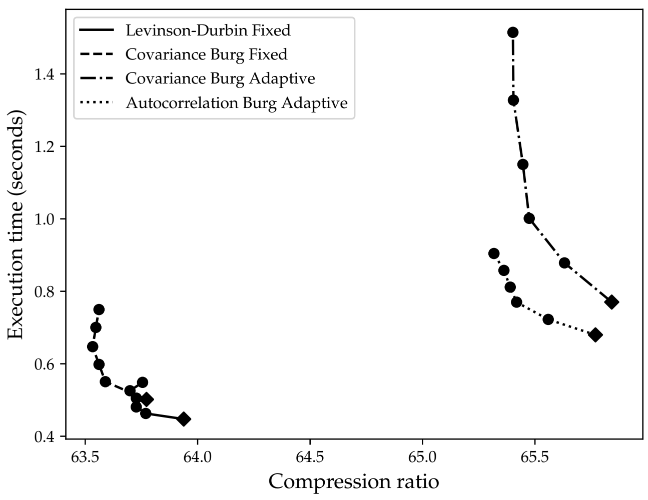


Figure 4.3: The compression ratio (compressed file size divided by original file size) and computation time of the four algorithm, for LPC orders 4, 6, 8, 10, 12 and 14, starting at the diamond and continuing along sequentially along the line. Uses the “Electronic Jazz” dataset from table 4.9.

Compression ratio and execution time for genre “Electronic”

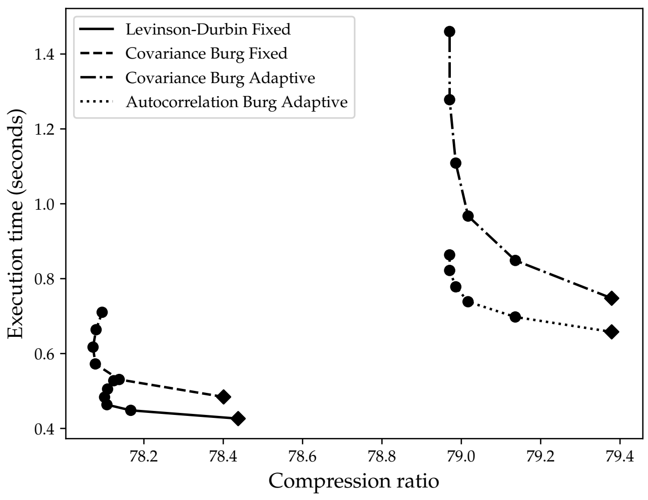


Figure 4.4: The compression ratio (compressed file size divided by original file size) and computation time of the four algorithm, for LPC orders 4, 6, 8, 10, 12 and 14, starting at the diamond and continuing along sequentially along the line. Uses the “Electronic” dataset from table 4.9.

Interestingly, these demonstrate the diminishing results when the order keeps increasing, as the for the Levinson-Durbin and Burg algorithms the larger orders actually increase the compression ratio.

Compression ratio and execution time for genre “Orchestral”

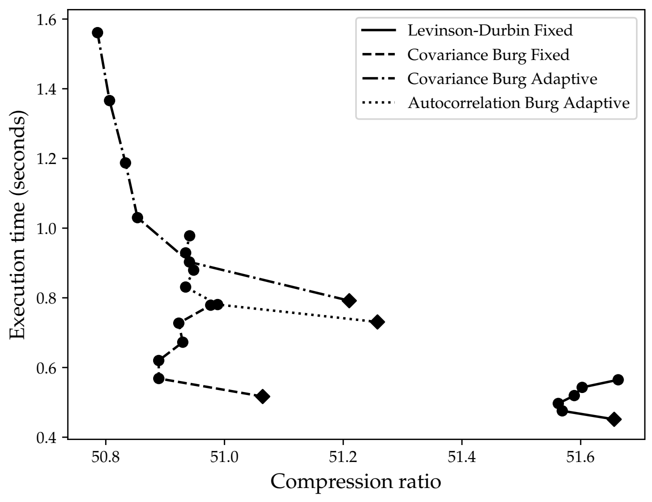


Figure 4.5: The compression ratio (compressed file size divided by original file size) and computation time of the four algorithm, for LPC orders 4, 6, 8, 10, 12 and 14, starting at the diamond and continuing along sequentially along the line. Uses the “Orchestral” dataset from table 4.9.

The results here are remarkable, as the compression ratios quickly increase as the order increases for all algorithm, except that this does not hold true for the covariance adaptive algorithm, and consequently it performs the best in terms of compression ratio.

We end our experiments with the genre for which the adaptive algorithms, especially the autocorrelation modification, perform by far the best.

Compression ratio and execution time for genre “Cuban”

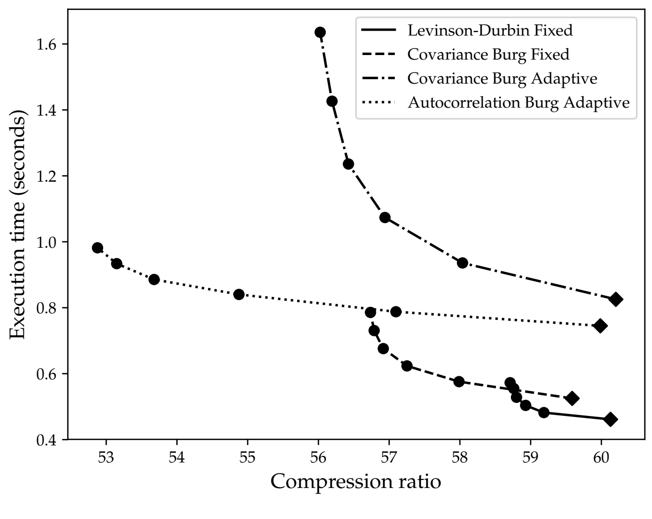


Figure 4.6: The compression ratio (compressed file size divided by original file size) and computation time of the four algorithm, for LPC orders 4, 6, 8, 10, 12 and 14, starting at the diamond and continuing along sequentially along the line. Uses the “Cuban” dataset from table 4.9.

4.3 Discussion

Throughout the previous section, the experiments showed that the adaptive algorithms can in some scenarios improve the compression rates compared to the standard fixed-blocksize algorithms, and that there seemed to be correlation between genre of music and effectiveness. Overall, improvements from the standard algorithms, if present, where not always convincing, and additionally, far from universal. Even though the ”Piano” and ”Cuban” genres showed favorable results for instance, for the electronic genres the adaptive algorithms performed poorly. The exact root cause for this difference, if it could be explained in terms of the musical content of the genres, is unclear as off yet. Moreover, we found the autocorrelation adaptive algorithm to do better than the standard adaptive algorithm, even though based on the results from section 2.5 we would expect it to do worse. The reason for this likely lies within the effectiveness of the blocksize detection algorithm 2.31, however this too requires further study.

Computationally speaking, the adaptive algorithms is slower in all scenarios, but the autocorrelation modification shows similar scaling in the order p as the fixed-blocksize methods. This is promising, since further optimisation could put this algorithm in a position to compete with the fixed-blocksize methods.

An interesting find separate from the adaptive algorithms is the fact that the Burg algorithm did better than the Levinson-Durbin algorithm for all genres we tested, even though the Levinson-Durbin algorithm is standard in most applications. However, the Levinson-Durbin algorithm consistently showed the fastest compression times.

In general, one would expect the larger and more suitable blocksizes the adaptive methods provide to improve the compression rates at least somewhat, if not significantly. This is not the case, and therefore we will take some time to examine possible reasons for this unfortunate result. We will discuss two important issues here, which are both possible explanations but need further investigation and study to assess their significance.

First, blocksize detection is suboptimal. This is motivated by the superiority of the autocorrelation adaptive Burg algorithm over the standard adaptive Burg algorithm as well. The detection is sensitive to noise and suffers whenever block edges are not clear, as we discussed in section 2.5 as well. Possible improvements are to smooth the error graph during the minimisation process to try to remove some of the impact of noise, or to partition large blocks into smaller blocks for whenever the detection algorithm has skipped over a block edge. Moreover, entirely different and more sophisticated minimisation algorithms could be employed instead of the described approach.

Secondly, we should ask ourselves whether our error criterion is even suitable for our purposes. We consider the squared residual error, even though the 2-norm has no apparent connection to the residual encoding mechanism of the Golomb-Rice codes. In fact, the 2-norm allows for relatively large outliers in the data, which can have severe negative consequences for the effectiveness of the Golomb-Rices codes. We can ask if a better error criterion might exist, and if so, if this error criterion can be integrated into the described adaptive algorithms.

4.4 Further enhancements

In this section we provide possible ideas which could enhance both the adaptive algorithms and the lossless compression format in general.

To begin, the presented adaptive algorithms have quite some room for improvements. Most importantly, a proper analysis of all detection parameters —the LPC order used for detection, N_{counter} and the allowed error range r— is certainly appropriate, as the values used throughout the experiments have been based on ad hoc and imprecise methods. Apart from this, many functional enhancements are possible as well. For instance, block size detection could take into account the rhythm of the music, preferring blocksizes which are close to (multiples of) previously found blocksizes. This could both stabilize the blocksize detection and serve as a basis for a relatively efficient rhythm detection algorithm. Additionally, by the cyclic nature of the error estimation, a lot of data is continuously discarded. Perhaps the previously computed values for k_m could be used to give more insight about the stationarity of the signal during the minimisation process.

We will end this discussion with a short comment about *streaming* versus *global* compression. The streaming compliance which the discussed techniques all satisfy is practical in many scenarios, however it might be limiting in some regards as well. Most audio signals have some form of repetition ingrained, however current compressors do not exploit this at all. The best example of this is in the MPEG-4 ALS format, which has a form of a *long term* predictor in addition to the short term linear prediction model [16], although streaming compliance is maintained. In any case, abandoning the streaming compliance —possibly partially— and focussing on global compression could lead to further improvements. Since knowing the structure of a song is especially essential in this case, efficient adaptive blocksize detection could play a role in further improving these techniques.

5 Conclusion

In this thesis we described the compression algorithm used by the FLAC format in detail, and introduced two new algorithms which improve on the standard fixed-blocksize signal partitioning by the adaptive selection of blocksizes in accordance with the audio signal in question. With the description of FLAC we provided an introduction to virtually all other current lossless compression codecs, as they share the same fundamental techniques, utilizing signal partitioning, linear prediction and entropy coding.

In chapter 2 we discussed the standard linear prediction algorithms and presented the two adaptive approaches: the adaptive covariance Burg algorithm and the autocorrelation adaptive Burg algorithm. The first degraded the complexity of the standard algorithms from $\mathcal{O}(pN)$ to $\mathcal{O}(p^2N)$, however the $\mathcal{O}(pN)$ complexity could be recovered with the modification adapted in the second. Both adaptive algorithms produced blocksizes that matched the audio adequately, however room for improvement is abundant. In chapter 4 all four algorithms were integrated into the FLAC format and compared, showing the new approaches overtake the standard approaches in terms of compression ratio for some genres, while lacking behind in others. Remarkably, the autocorrelation adaptive Burg algorithm improved on the compression ratio of the covariance adaptive Burg algorithm. Compression time is worse than the standard algorithms, but comparable. Especially the autocorrelation adaptive Burg algorithm, which scaled conform its $\mathcal{O}(pN)$ complexity, is promising in this regard.

Additionally, in chapter 2 we showed the LPC residuals for all algorithms tend to conform to the Laplacian distribution rather than the typical normal distribution. We highlighted the Golomb-Rice codes, a particular class of universal symbol codes, in chapter 3 and proved their optimality for encoding residuals of the 2-sided geometric distribution, the discrete counterpart to the aforementioned Laplacian distribution.

Overall, we affirm that the adaptive algorithms could lead to improvements over the existing approaches. Further development is needed, with special focus on the improvement of the blocksize detection, and perhaps a reevaluation of the used error measure is in order, as described in 4.3. Next, the algorithms should be integrated better in the FLAC format, using all features it presents. In fact, integration of these algorithms in the recent improvements over FLAC, the MPEG-4 ALS [16] and IEEE Advanced Audio Coding [12] formats, is possible and could possibly lead to further improvements in the field.

Populaire samenvatting

Digitale media, zoals foto-, video- en audiobestanden, bevatten zeer veel informatie, en het is dan ook geen verrassing dat ze al snel veel opslagruimte kunnen opeisen. Met *compressie* trachten we dit probleem te verhelpen. Door slim gebruik te maken van de structuur van de data proberen we een groot deel van de overtollige gegevens uit het bestand te verwijderen, zodat we de foto, video of audio intact blijft, maar waarbij aanzienlijk minder opslagruimte noodzakelijk is. Voorbeelden van gecomprimeerde bestanden zijn bestanden met de extensies `.jpg` en `.png` voor afbeeldingen, `.mp3` en `.flac` voor audio, `.mp4` voor video en `.zip` voor algemene datacompressie.

Compressie kunnen we opdelen in twee categorieën: *lossy* compressie en *lossless* compressie. De eerste staat bij het comprimeren toe dat enkele relatief onbelangrijke informatie uit het bestand wordt weggegooid, om zo nóg kleinere bestandsgroottes the behalen. Bij de tweede mag er juist geen informatie verloren gaan. Het originele bestand is daarom compleet terug op te halen, maar het uiteindelijke bestand zal hierdoor groter zijn dan wat met lossy compressie behaald kan worden.

In deze scriptie ligt de focus op lossless audiocompressie. Het FLAC bestandstype is het meest gebruikte bestandstype voor dit doeleinde. Het welbekende MP3-formaat is daarentegen een lossy audiocompressieformaat. Voor het beluisteren van muziek is lossless eigenlijk niet noodzakelijk —met MP3 kunnen bestanden doorgaans vijf keer kleiner worden gemaakt dan met FLAC— en het verschil is nauwelijks hoorbaar. Lossless compressie wordt daarom vooral gebruikt bij muziekproductie, digitale muziekarchieven en wetenschappelijke of medische doeleinden.

FLAC maakt hierbij gebruik van een compressie-algoritme dat ook de basis is voor bijna alle huidige lossless-compressieformaten. Dit algoritme werkt via de volgende drie stappen.

Stap 1: Het opdelen van het audiosignaal
Om te beginnen wordt het audiosignaal opgedeeld in *blokken*, korte aaneensluitende fragmenten geluid, die daarna allemaal apart gecomprimeerd zullen worden. Alle huidige formaten, zoals ook FLAC, kiezen hierbij blokken met een vaste grootte van bijvoorbeeld één zesde seconde.

Stap 2: Het modelleren van de audio
De audio in elk blok wordt vervolgens zo goed mogelijk beschreven door een bepaald *model*, zodat met alleen de modelparameters het belangrijkste geluid in het blok al opgeslagen kan worden. Het model dat hierbij wordt gebruikt heet *Linear Predictive Coding*, en voor het achterhalen van de beste modelparameters bestaan verschillende algoritmes.

Dit model blijkt heel goed te werken voor audio waarbij de frequentie-inhoud niet veranderd. Vandaar dat we de audio in stap 1 opdeelde! Doordat de blokken klein zijn bevat elk blok namelijk met grote kans maar één akkoord of noot.

Stap 3: Het efficiënt opslaan van de modelfout
Het model zal helaas nooit 100% accuraat zijn, en we willen juist dat we helemaal geen informatie verliezen! Daarom geven we naast de modelparameters ook de fout van het model mee in het bestand. De laatste stap is hierdoor het efficiënt opslaan van deze modelfout, wat in FLAC gedaan wordt met de zogeheten *Golomb-Rice entropy codes*.

Hiermee kan het bestand worden opgeslagen als de modelparameters en modelfout voor elk blok, waarmee het origine audiobestand compleet mee kan worden opgebouwd. Het plaatje op het voorblad geeft een samenvatting van dit stappenplan, waarbij elk horizontale gedeelte overeenkomt met één van de bovenstaande drie stappen.

Naast het beschrijven van de standaardmethodes voor het toepassen van dit algoritme heb ik ook gekeken naar een mogelijke verbetering. In stap 1 beschreef ik hoe blokken normaal van vaste grootte worden gekozen, maar deze keuze zal in de praktijk bijna nooit daadwerkelijk met de akkoorden en het ritme van de muziek overeenkomen. Hierdoor gemotiveerd beschrijf ik een algoritme dat tijdens de modelleerstap ook blokgroottes probeert te kiezen in overeenstemming met de structuur van het liedje.

Dit nieuwe algoritme blijkt redelijk goed te werken, en de gekozen blokgroottes komen inderdaad overeen met de geanalyseerde muziek, maar er valt ook zeker nog veel te verbeteren. Zo worden blokken niet altijd op de beste manier gekozen, waardoor de uiteindelijke compressie slechts voor sommige muziekgenres beter is dan de standaard algoritmes, en voor andere juist slechter. Maar met de vele mogelijke uitbreidingen, lijkt het nieuwe algoritme toch veelbelovend te zijn!

References

[1] *FLAC format documentation.* https://xiph.org/flac/format.html. Xiph.Org Foundation, Accessed: June 2020.

[2] K. Brandenburg, *MP3 and AAC explained*, Fraunhofer Institute for Integrated Circuits, (2001).

[3] J. P. Burg, *Maximum entropy spectral analysis*, PhD thesis, Stanford University, May 1975.

[4] J. Coalson, *Free lossless audio codec.* https://tools.ietf.org/html/draft-xiph-cellar-flac-00, June 2017. Accessed: June 2020.

[5] C. Collomb, *Linear prediction and levinson-durbin algorithm*, (2019).

[6] T. M. Cover and J. Thomas, *Elements of information theory*, John Wiley & Sons, New York, second ed., 2006.

[7] E. de Castro Lopo, *FLAC reference encoder.* https://github.com/xiph/flac/, 2011. Accessed: June 2020.

[8] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, *The use of asymmetric numerical systems as an accurate replacement for Huffman coding*, 2015 Picture Coding Symposium (PCS), (2015), pp. 65–69.

[9] R. W. Freund, *A look-ahead Bareiss algorithm for general Toeplitz matrices*, Numerische Mathematik, 68 (1994), pp. 35–69.

[10] S. Golomb, *Run-length encodings (corresp.)*, IEEE Transactions on Information Theory, 12 (1966), pp. 399–401.

[11] R. M. Gray, *A history of realtime digital speech on packet networks: Part II of linear predictive coding and the internet protocol*, Foundations and Trends in Signal Processing, 3 (2010), pp. 203–303.

[12] H. Huang, H. Shu, and R. Yu, *Lossless audio compression in the new IEEE standard for advanced audio coding*, 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), (2014), pp. 6934–6938.

[13] F. Itakura, *Minimum prediction residual principle applied to speech recognition*, IEEE Transactions on Acoustics, Speech, and Signal Processing, 23 (1975), pp. 67–72.

[14] F. Keiler, D. Arfib, and U. Zölzer, *Efficient linear prediction for digital audio effects*, Proceedings of the COST G-6 Conference on Digital Audio Effects, (2000).

[15] A. Kiely and M. Klimesh, *Generalized Golomb codes and adaptive coding of wavelet-transformed image subbands*, The Interplanery Network Progress Report, (2003).

[16] T. Liebchen, T. Moriya, N. Harada, Y. Kamamoto, and Y. A. Reznik, *The MPEG-4 audio lossless coding (ALS)*, Audio Engineering Society, (2005).

[17] H. D. Luke, *The origins of the sampling theorem*, IEEE Communications Magazine, 37 (1999), p. 106–108.

[18] J. Makhoul, *Linear prediction: A tutorial review*, Proceedings of the IEEE, 63 (1975), pp. 561–580.

[19] J. Makhoul, *Stable and efficient lattice methods for linear prediction*, IEEE Transactions on Acoustics, Speech, and Signal Processing, 25 (1977), pp. 423–428.

[20] J. Makhoul, S. Roucos, and H. Gish, *Vector quantization in speech coding*, Proceedings of the IEEE, 73 (1985), pp. 1551–1588.

[21] D. O'Shaughnessy, *Linear predictive coding, one popular technique of analyzing certain physical signals*, IEEE Potentials, (1988).

[22] K. M. M. Prabhu, *Window functions and their applications in signal processing*, CRC Press, first ed., 2017.

[23] Y. A. Reznik, *Coding of prediction residual in MPEG-4 standard for lossless audio coding (MPEG-4 ALS)*, 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, (2004).

[24] T. Robinson, *SHORTEN: Simple lossless and near-lossles waveform compression*, (1994).

[25] W. Rudin, *Real and Complex Analysis*, McGraw-Hill Book Company, third ed., 1987.

[26] G. J. Sullivan, P. Topiwala, and A. Luthra, *The H.264/AVC advanced video coding standard: Overview and introduction to the fidelity range extensions*, SPIE Conference on Applications of Digital Image Processing XXVII, (2004).

[27] L. N. Trefethen and D. Bau III, *Numerical Linear Algebra*, Society for Industrial and Applied Mathematics, first ed., 2017.

[28] R. Yu, X. Lin, S. Rahardja, R. Geiger, J. Herre, and H. Huang., *MPEG-4 scalable to lossless audio coding*, 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, (2004).

Appendix

A Vector quantization

The idea of vector quantization is to represent the LPC parameters in a compact and transferable manner. We give a brief introduction to the quantization used by FLAC here, and for more information we refer to [20].

Two parameters of importance are the *precision* P and *shift value* s. The idea is to represent the parameters a_j by a signed two's complement integer value $\hat{a}_j \in \mathbb{Z}$ of P bits long, such that $a_j \approx \hat{a}_j >> s$. The precision P is often set beforehand, typically to 12, while the best shift is determined as presented in the following algorithm. Afterwards, the values $\hat{a}_j$ are computed sequentially, while taking into account the *quantization error* ε of the previously computed quantized values. Note that since bitshifting is not defined for the real coefficients a_j, we instead compute the functionally equivalent $a_j \cdot 2^s$.

Algorithm (Vector quantization)

```
1: procedure VectorQuantization
2:     $a_{\max} \leftarrow \max_{1\le j\le p} \{|a_j|\}$
3:     $s \leftarrow P - (\lfloor \log(a_{\max}) \rfloor + 1)$
4:     $\varepsilon \leftarrow 0$
5:     for $1 \le j \le p$ do
6:         $\varepsilon \leftarrow \varepsilon + a_j \cdot 2^s$
7:         $\hat{a}_j \leftarrow \mathrm{round}(\varepsilon)$
8:         if $\hat{a}_j$ does not fit in $P$ bits then
9:             Discard all bits from $\hat{a}_j$ with higher order than $P$
10:        end if
11:        $\varepsilon \leftarrow \varepsilon - \hat{a}_j$
12:    end for
13: end procedure
```

The residuals can then be computed by

$$u[n] = s[n] - \Big(\sum_{j=1}^{p} s[n-j] \cdot \hat{a}_j\Big) >> s.$$

The described coefficient quantization, which is employed by FLAC, is straightforward. Quantization can however have a huge impact on the model being quantized, and for this reason LPC quantization is of special importance for non-lossless applications such as low-bitrate speech encoding, which do not send along the residuals and are therefore more susceptible to disturbances in the coefficient set. Common representation are the *log area ratio* and *line spectral pairs* quantization methods, although for our purposes the above quantization is sufficient.

B Cyclic Redundancy Checks

Cyclic Redundancy Checks (CRC's) are a type of error-checking code, designed to detect whether transmission of a data has been successful. That is, that no bits have been accidentally flipped during communication over noisy network connections. They achieve this by appending a *check* bitstring c to the message, computed uniquely from the same message. CRC's are popular since they are particularly effective at error detection and easy to implement in hardware. Many small variations on the basic principle of CRC's exist, which is why we provide the pseudo-code for the version used by FLAC here.

CRC's calculate the check bitstring c by way of polynomial division over the field of $\mathbb{F}_2$. Given a *generator polynomial* P, the binary message $m = b_{n-1}b_{n-2}\cdots b_0$ is interpreted as a polynomial

$$M(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_1x^1 + b_0x^0 \in \mathbb{F}_2[x],$$

and the remainder $C(x)$ of $M(x)/P(x)$ is computed. Then c is set to equal the bitstring associated with C in the same way m can be retrieved from M. For CRC-8, P has degree 9 such that c is 8 bits long. Similarly, for CRC-16, P is of degree 17. We will denote ℓ for the length of c.

Calculation of the remainder C is done iteratively using long division, by setting $C \leftarrow M$ and iterating over C from c_{n-1} to c_0, subtracting $P(x)x^{i-\ell-1}$ from $C(x)$ whenever $c_i = 1$. The advantage of using $\mathbb{F}_2$ as the underlying field is that the complete division can be done efficiently using bitwise operations. We can confine c to just ℓ bits, iterating over C by repeatedly bitshifting c to the left by 1 as well as shifting in a new value from m. When the most significant bit of c is equal to 1, the next iteration requires a subtraction with P, which in $\mathbb{F}_2[x]$ is the same as the XOR operation $\otimes$ with the bitstring p associated with P.

Algorithm (CRC computation)

```
1:  procedure CRC(Message $m = b_{n-1}b_{n-2}\cdots b_0$, Gen. polynomial $p$)
2:      $c \leftarrow 0$
3:      for $i = n-1, n-2, \ldots, 0$ do
4:          if the most significant bit of $c$ is 1 then
5:              $c \leftarrow (c$ << $1) + b_i$
6:              $c \leftarrow c \otimes p$
7:          else
8:              $c \leftarrow (c$ << $1) + b_i$
9:          end if
10:     end for
11:     return $c$
12: end procedure
```

Often, c is referred to as the *shift register*. Further improvements can be made by shifting in the bits from m in ℓ long chunks, which is why ℓ is generally multiple of 8, however we omit those here.

C UTF-8 character encoding

The 8-bit Unicode Transformation Format (UTF-8) encoding standard was originally developed for efficient encoding of the complete Unicode character set. It uses *variable-width* encoding, where characters with larger Unicode code points would be assigned a larger numbers of bytes, while character with small code points, such as the Latin alphabet, are encoded with only a single byte (in fact, the Latin code points coincide with ASCII).

Of course, this encoding scheme is nothing more than the encoding of the integer code points associated with the characters, and FLAC reuses this encoding method for plain integer values.

The encoding of a binary integer n with UTF-8 comes in two parts: the first byte and all remaining bytes. The first byte encodes the total amount of bytes plus 1 in unary — terminated by a `0`— together with the highest order bits of n that fit, while the remaining bytes contain the rest of n in binary. Each non-first byte is made up of the bitstring `10` followed by a 6 bit long *field* which contains part of n. Thus, the amount of fields we need is equal to the amount of bits required to write n divided by 6, such that

$$f \leftarrow \left\lfloor \frac{\lfloor \log(n) \rfloor + 1}{6} \right\rfloor + 1.$$

If $\big(\lfloor \log(n) \rfloor + 1\big) \bmod 6 < 8 - (f+1)$ we know that we can fit the first field in the first byte, after the unary encoded $f-1$ plus 1, which would then take up exactly $(f-1)+2 = f+1$ bits.

Finally, there is the special case of ASCII characters for which only the first byte is used, and the first bit is set to 0. We also note that the first byte can never have the prefix `10`, which enables decoders to start at arbitrary points in a character stream. Writing `&` for the bitmask operation, we arrive at the following encoder.

Algorithm (UTF-8)

```
procedure WriteUTF-8(Integer $n$)
    if $n < 128$ then
        Write $n$ in 8 bits
    else
        #bits = $\lfloor \log(n) \rfloor + 1$
        $f \leftarrow \lfloor \#\text{bits}/6 \rfloor + 1$
        if #bits mod $6 < 8 - (f+1)$ then
            $f \leftarrow f - 1$
        end if
        Write $\texttt{1}^{f+1}\texttt{0}$ and then $n$ >> $6 \cdot f$ in $6 - f$ bits
        for $i = 1, 2, \ldots, f$ do
            Write 10 and then 111111 & $\big(n$ >> $6 \cdot (f - i)\big)$
        end for
    end if
end procedure
```